1984-06

Application of a silicon complier to VLSI design of digital pipelined multipliers

Carlson, Dennis J.

http://hdl.handle.net/10945/19164

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

APPLICATION OF A SILICON COMPILER TO VLSI DESIGN OF DIGITAL PIPELINED MULTIPLIERS

by

Dennis J. Carlson

June 1984

Thesis Advisor: D. E. Kirk

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dennis J. Carlson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE June 1984
		13. NUMBER OF PAGES 141
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

VLSI Design, MacPitts, Pipelined Multipliers, Silicon Compiler, CAD Tools

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The concept and application of silicon compilers is described. The process of employing the MacPitts silicon compiler to design an 8-bit pipelined digital multiplier is presented, and the resulting design is evaluated. The process of installing and debugging the MacPitts compiler and the Caesar VLSI graphics editor on the VAX 11/780 computing facilities at NPS is documented in appendices.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers

by

Dennis J. Carlson
Lieutenant Commander, United States Navy
B.S., Rensselaer Polytechnic Institute, 1969

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

The concept and application of silicon compilers is described. The process of employing the MacPitts silicon compiler to design an 8-bit pipelined digital multiplier is presented, and the resulting design is evaluated. The process of installing and debugging the MacPitts Compiler and the Caesar VLSI graphics editor on the VAX-11/780 computing facilities at NPS is documented in appendices.

TABLE OF CONTENTS

I. INTRODUCTION 11
  A. BACKGROUND 11
  B. CURRENT RESEARCH GOALS 12

II. APPROACHES TO SILICON COMPILATION 15
  A. VLSI DESIGN ACTIVITIES DOMAIN 15
  B. EVALUATION CAVEAT 17
  C. LIMITED SPECTRUM COMPILERS (TRANSLATORS) 18
  D. BROAD-SPECTRUM SILICON COMPILERS 19
    1. Floor Planners 19
    2. Behavioral Specification Compilers 21

III. USING MACPITTS 30
  A. THE INPUT FILE 30
    1. Fundamentals of the MacPitts Language 30
    2. Two Multiplier Examples 36
  B. INVOCATION OPTIONS 42
  C. USE OF THE MACPITTS INTERPRETER 46
  D. EVOLUTION OF THE 8 BIT PIPELINED MULTIPLIER 49
    1. Design Motivation and Constraints 49
    2. First Design: 3 Stages, 8 Bits on One Chip 50
    3. First Partitioning: 2 Bits, 1 Stage Pipeline 51
    4. Second Partitioning: 4 Bits, 2 Stage Pipeline 54
    5. Third Partitioning: 2 Bits, 4 Stage Pipeline 57

E. DESIGN VALIDATION 60
1. Functional Simulation 60
2. Design Rule Checking 63
3. Node Extraction and Event Simulation . . . 66
F. SUMMARY OF ACTIVITIES IN THE MACPITTS DESIGN CYCLE 68

IV. MACPITTS PERFORMANCE 72
A. LAYOUT ERRORS AND INEFFICIENCIES 72
1. Inefficiencies 72
2. Errors 75
B. ORGANELLES VS. STANDARD CELLS 77
C. SOFTWARE INCOMPATIBILITIES 78

V. CONCLUSION 79
A. SUMMARY 79
B. RECOMMENDATIONS 80

APPENDIX A: INSTALLATION OF MACPITTS ON VAX-11/780 UNDER UNIX 4.1 AND 4.2 81
A. INSTALLATION UNDER UNIX 4.1 OPERATING SYSTEM 81
B. INSTALLATION UNDER UNIX 4.2 OPERATING SYSTEM 85

APPENDIX B: INSTALLATION OF THE CAESAR VLSI EDITOR UNDER UNIX 4.1 AND 4.2 88
A. INSTALLATION UNDER UNIX 4.1 88
B. INSTALLATION UNDER THE UNIX 4.2 OPERATING SYSTEM 89

APPENDIX C: MANUAL PAGES FOR BERKELEY DESIGN TOOLS . . . 91

APPENDIX D: SIMULATION RESULTS FOR MULTIP8C MULTIPLIER 122

APPENDIX E: LAYOUT PHOTOGRAPHS 131

LIST OF REFERENCES 137

BIBLIOGRAPHY 139

INITIAL DISTRIBUTION LIST 140

LIST OF TABLES

I. Statistics For MacPitts Multiplier Chip Designs 61
II. MacPitts Source Files 82

LIST OF FIGURES

2.1 VISI Design Activities Spectrum 16
2.2 Typical floor plan produced by the F.I.R.S.T. Silicon Compiler 20
2.3 Floor Plan of the MacPitts Target Architecture 23
2.4 MacPitts Register Circuit and Timing Diagram . . 25
2.5 MacPitts Program Data Flow 27
3.1 Multic.mac Source File 37
3.2 Example of the Multic Behavioral Specification 40
3.3 Multip.mac Source file 41
3.4 Compiler Statistics for multip 43
3.5 A MacPitts Interpreter Session for multip . . . 48
3.6 Multip8.mac Source File 52
3.7 Multip8.mac Source File (Continued) 53
3.8 Use of Ports and Registers in multip8.mac . . . 54
3.9 Data Path Architecture of Multip8 Chip 55
3.10 Block Diagram of First Partitioning 56
3.11 Multip8a.mac Source File 57
3.12 Multip8b.mac Source File 58
3.13 Multip8c.mac Source File 59
3.14 Values: Program to Compute Multip8c Output . . 64
3.15 Mextra .log File for Mul8c.cif. 66
3.16 Two Macro Driver Files for Event Simulation . . 67
4.1 Data Path Ouput Routing 74
D.1 Macpitts Interpreter Results 122
D.2 MacPitts Interpreter Results, (continued) . . 123
D.3 MacPitts Interpreter Results, (Continued) . . 124

D.4 MacPitts Interpreter Results, (Continued) . . 125
D.5 MacPitts Interpreter Results, (Continued) . . 126
D.6 MacPitts Interpreter Results, (Continued) . . 127
D.7 Event Simulation Results 128
D.8 Event Simulation Results, (Continued) 129
D.9 Event Simulation Results, (Continued) 130
E.1 multic (top), multip (bot) 132
E.2 multip8 (top), multip8a (bot) 133
E.3 multip8b (top), multip8c5 (bot) 134
E.4 multip8c4 (top), multip8c4d (bot) 135
E.5 Layout Errors in kchip2 136

ACKNOWLEDGEMENTS

I would like to thank the following individuals for their assistance in the completion of this thesis:

Naval Postgraduate School

Dr. Donald Kirk
Prof. Robert Strum
Dr. Herschel Loomis
Mr. Al Wong

Massachusetts Institute of Technology
Lincoln Laboratory

Mr. Kenneth W. Crouch
Dr. Antun Domic

University of California at Berkeley

Dr. John K. Ousterhout
Dr. Keith Sklower

Stanford University

Dr. Robert Mathews
Ms. Susan Taylor

University of Kansas

Dr. Gerry L. Kelly

I. INTRODUCTION

A. BACKGROUND

The initial work done on the design of very large scale integrated circuits (VLSI) at the Naval Postgraduate School (NPS) used a set of software tools which require designer interaction at all levels of the design process. These tools and their use is described in a recent thesis by Conradi and Hauenstein [Ref. 1].

Their design approach centers around the use of: (1) machine-generated programmable logic arrays (PLA's) specified in a language which translates boolean equations into circuit layouts, and (2) a library of standard cell layouts from which other required circuit primitives are selected. The designer arranges the PLA's and standard cells on a "floorplan" designed by heuristic methods, and interconnects them with a network of individual wires devised by the designer and encoded as a "wirelist." The floorplan layout and addition of interconnecting wires must be done manually, typically on graph paper at the drawing board. The results are manually encoded in an input file format readable by a layout language program ("cll" in the case of the cited research) which merges the designer's floorplan and wirelist with: (1) the selected library cell layout descriptions and (2) the PLA layout descriptions produced by the separate PLA generation program. The circuit layout program then produces a description of the total design in another standard file interchange format, the Caltech Intermediate Form, (CIF) described by Mead and Conway [Ref. 2: pp. 115-127]. The CIF file can then be used as a source for extracting design validation information, as well for producing the photographic masks used for circuit fabrication.

The design process outlined has the advantage of giving the designer thorough control over the architecture of the circuit. The human ability to evaluate alternatives, recognize patterns and grasp complex multi-dimensional relationships between individual elements and the whole design exceeds that of any current machine algorithm.

On the other hand, this process absorbs large amounts of the designer's time in performing the drudgery of planning and encoding the layout details. There are at least four things wrong with involving the designer at this level:

(1) It is repetitious work, and therefore error-prone.

(2) It is slow. (Southard [Ref. 3] and others have noted that design costs far outweigh production costs for custom VLSI.)

(3) Preoccupation with mechanical details restricts a designer's freedom to explore high-level architectural issues such as bus structure, degree of pipelining, and speed-complexity tradeoffs.

(4) Major modifications to the layout are very expensive to make if they come late in the design cycle, i.e. after cell interconnection.

E. CURRENT RESEARCH GOALS

With this background for motivation, it was decided to investigate additional VLSI computer-aided design tools which would reduce time-to-design, minimize the occurrence of human error in layout, and make it possible to explore design alternatives with greater ease.

The major tool available in the VLSI research community for this purpose is MacPitts. MacPitts (the name is derived from two early researchers, McCulloch and Pitts who studied neurological systems from a mathematical and logic standpoint) is a silicon compiler developed at the Massachusetts

Institute of Technology's Lincoln Laboratories in 1981-1982 [Ref. 4]. A silicon compiler, according to one recent definition [Ref. 5] which captures current usage of this often misunderstood term, is "a program that, given a description of what a circuit is supposed to do, will produce a chip layout that implements that function in silicon." There is enough latitude to allow fundamentally different approaches to silicon compilation to coexist under this definition, as will be demonstrated in the following chapter. In any case, however, the term compiler is apt. Like software compilers, these programs take high-level source code descriptions which are human-readable (and perhaps, but not necessarily, algorithmic) and "convert" them into low-level object code (a CIF file) which is directly readable by a machine. In the case of a silicon compiler, however, the machine is not a general-purpose computer, but a photo-resist mask generator at a silicon foundry facility that fabricates integrated circuits.

Another function that the most advanced silicon compilers perform is resource allocation. Software compilers free the programmer from making decisions on where in available memory space to store a particular machine code word. Silicon compilers, at their best, free the designer from deciding where on available silicon area to place a particular circuit element. Resource allocation is a one-dimensional job in software compilers, but a two-dimensional job in silicon compilers. The constraints on efficient resource allocation in silicon are severe--compactness is almost always one goal, as is speed of operation (minimum propagation delay.) In memory allocation, compactness is not essential, unless one is using a sequential access memory.

Installation of MacPitts on the NPS VAX-11/780 computer facility was expected to be a "turn-key" operation. This was in fact not the case. A large amount of effort was

spent in researching and performing the modifications to the host computer environment which enable it to run the MacPitts system, as well as in troubleshooting the distributed MacPitts source code itself. The installation process is described in Appendix A.

MacPitts has no progressive breakpoint facilities to allow a designer freedom to observe or alter the layout process at any point during execution. Once invoked, MacPitts produces a final interconnected layout, complete with bonding pads, or no layout at all. Therefore, it was considered worthwhile to implement the color graphics editor, Caesar, designed by John Ousterhout at the University of California at Berkeley [Ref. 6]. This tool allows the chip layout to be examined in detail on a color CRT monitor, and permits editing of the layout. Caesar represents the layout internally as a hierarchy of cells, which yields insight into the ways that MacPitts partitions the layout process.

The installation of Caesar, while not as difficult as MacPitts, involved setting some site-dependent parameters as well as finding and correcting a bug in the distributed source code. These activities are described in Appendix B. Appendix C contains a copy of the on-line manual pages for Caesar and other Berkeley tools used in this research.

II. APPROACHES TO SILICON COMPILATION

A. VLSI DESIGN ACTIVITIES DOMAIN

When trying to understand how silicon compilers work it is instructive to think of two design problems in the order in which they must be attacked. The first is translation of a brief behavioral or functional description into a more precise intermediate description that is still independent of the specific implementation technology. The second is the automatic generation of a chip layout in a target semiconductor medium, using the intermediate description as a guide. It is important to separate the second activity from the first when one is designing a silicon compiler because of the speed at which the target semiconductor technologies are evolving. That is, complementary metal oxide semiconductor (CMOS) processes are rapidly overtaking N-channel metal oxide semiconductor (NMOS) processes. Multiple-layer metalization is also becoming more common, and minimum circuit feature sizes are shrinking as better control over the manufacturing processes is achieved. Computer architectures and functions evolve more slowly, by comparison.

These two problems may be further subdivided. Werner [Ref. 7] has contributed the idea that a spectrum of VLSI design activities exists with corresponding media for the exchange of information by the computer-aided design tools employed at each band in the spectrum. (See figure 2.1.) Silicon compilers try to span the whole spectrum, an ambitious undertaking.

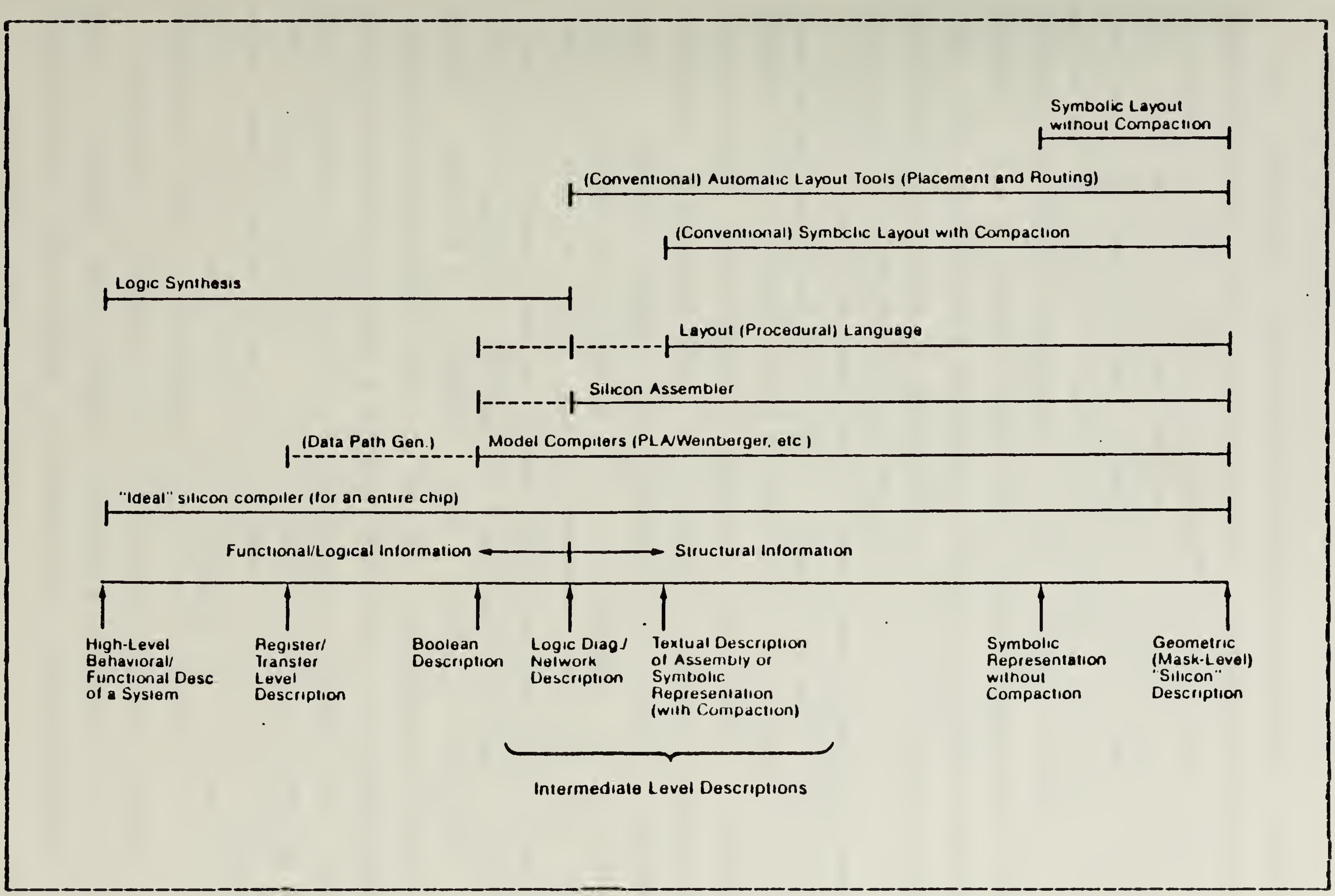


Figure 2.1 VLSI Design Activities Spectrum [Ref. 7].

B. EVALUATION CAVEAT

It should be recognized that all silicon compilers designed to date have to some extent traded performance of the ultimate VLSI design (as measured by operating speed and area efficiency) for reduced design time for the chip (and for the silicon compiler itself.) Gross [Ref. 8] quotes estimates for reduced design costs (time) by use of broad spectrum silicon compilers to be a factor of 20. But Wallich, in a recent survey of silicon compiler efforts [Ref. 5], states that designs produced by silicon compilers available today tend to range from 15 to 200 percent larger than equivalent hand-crafted designs.

Still, silicon compilers have been misunderstood by researchers as noted by Gross. Some, without fully understanding the dimensionality of the VLSI design process, believe that the design problem can be almost completely solved by the application of current software methods and tools. Others, seeing the obvious limitations of contemporary silicon compilers and not grasping the potential contributions to VLSI from computer science technology transfer, believe that efficient VLSI designs will always be essentially manual. Murphy of Bell Laboratories, quoted by Werner [Ref. 7], states that "total automation is inappropriate--either now or in the foreseeable future--in anything where you have a competitive need for performance." Nevertheless, Bell Labs is conducting research of its own into silicon compilers. Their "Plex" project reported in a more recent paper [Ref. 9] produces layouts of microcomputers given, as input, the <u>program</u> (in assembly or C language) that the microcomputer is to execute.

According to Wallich, the ultimate silicon compiler, now just a dream, will not only be able to take a behavioral description and produce a geometrical description of the

chip suitable for input to a mask making machine, but will do so for any kind of chip--microprocessor, signal processor, or even analog-digital hybrid for which the design rules are far more complex. The subtle process of architectural optimization (i. e. selecting a best floor plan from the myriad possibilities,) which occurs in the middle of the design activities spectrum, has so far not been captured in an algorithm. To achieve some breadth without being overwhelmed by complexity, silicon compilers have tended to contain built-in assumptions about a "target architecture." They are optimized for producing a certain class of circuits--mostly microprocessors--and produce layouts of reasonable area and speed only for applications best suited to their target architecture.

C. LIMITED SPECTRUM COMPILERS (TRANSLATORS)

For completeness, it is necessary to mention those VLSI design tools in current use which fall short of covering the design spectrum. They are:

- Random logic/Standard-cell place-and-route systems,
- Module compilers to implement boolean logic, including:
    - Gate array compilers,
    - PLA generators,
    - Regular expression compilers for finite-state machines,
- Layout Languages,
- Interactive graphical layout editors.

D. BROAD-SPECTRUM SILICON COMPILERS

1. Floor Planners

a. Common Properties

The first broad spectrum translators of interest are the floor planners. They all employ a structural specification language in which the specification always corresponds extremely closely to a description of the designer's mental model of how the chip should be laid out. They produce, as an initial output, a skeleton of the layout similar to an architect's floor plan. Subsequently, floor planners fill the "rooms" with cells from a standard library. Some floor planners, of which Johannsen's Bristle Blocks is a pioneering example [Ref. 10], can linearly stretch cells to match up the interconnections of abutting cells (so-called "pitch matching.")

b. F.I.R.S.T.

The current state of the art in floor planners is represented by the F.I.R.S.T. (Fast Implementation of Real-Time Signal Transforms) silicon compiler developed at Edinburgh University [Ref. 11]. The F.I.R.S.T. compiler produces layouts of digital signal processing systems implemented as hard-wired networks of pipelined bit-serial operators. The floor plan of F.I.R.S.T. chips (see figure 2.2) consists of a central wiring channel with operators arranged as function blocks around the "waterfront." Each bit-serial operator is implemented as a separate function block which in turn is assembled from a library of hand-designed cells. The function blocks are arranged, in the order of their high-level specification by the designer, in two rows along either side of the wiring channel which accommodates all interconnections between the blocks. This uncomplicated and

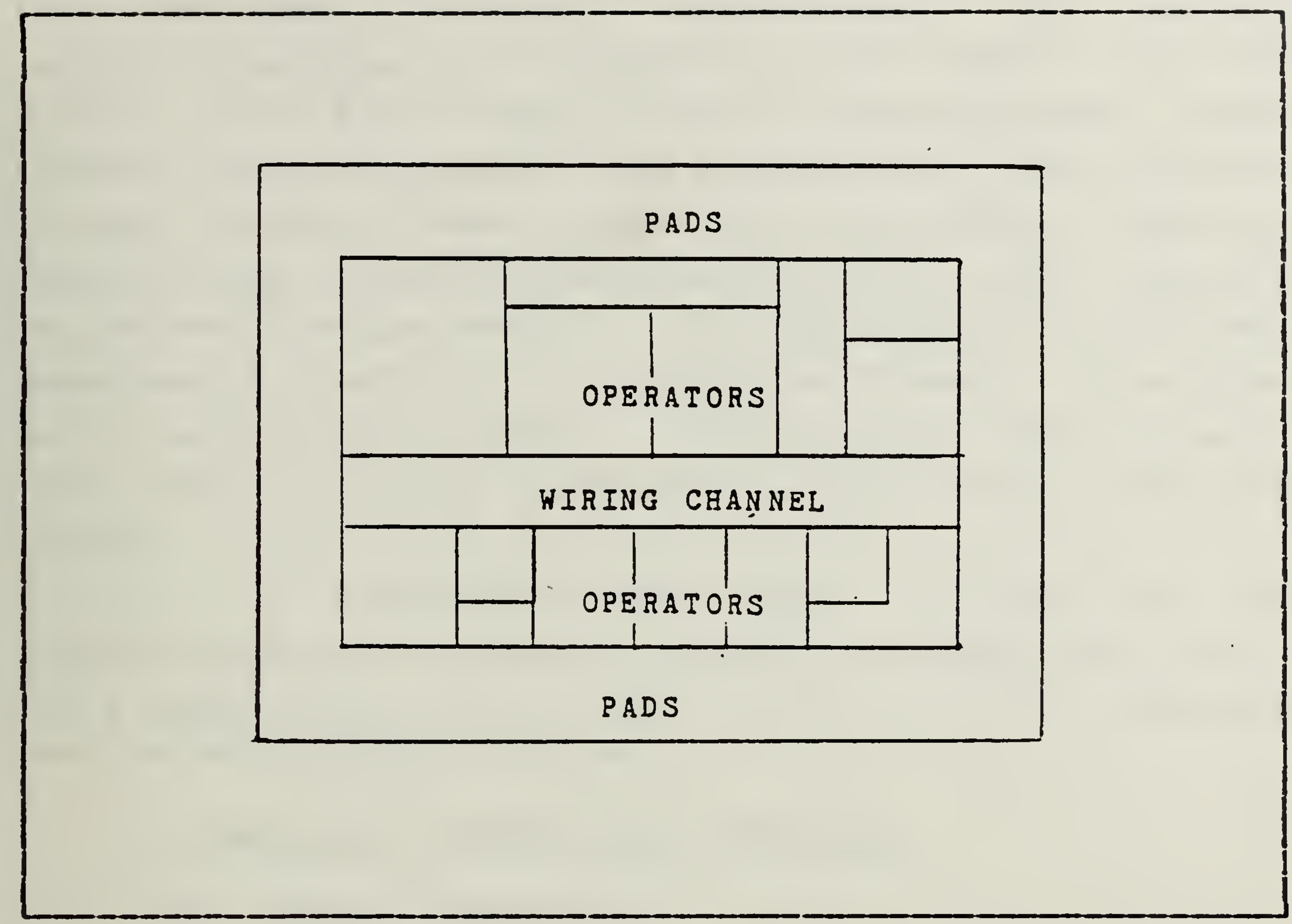


Figure 2.2 Typical floor plan produced by the F.I.R.S.T. Silicon Compiler.

novel layout methodology results in the non-use of about 20% of the total chip area (because the blocks may have varied heights.) At present, F.I.R.S.T. supports only the N-channel metal oxide semiconductor (NMOS) technology.

The F.I.R.S.T. software consists of a small suite of programs which provides the designer with a complete specialized design environment. At the top level is a language compiler that accepts a structural description of the circuit in terms of a net list of bit-serial operators. The F.I.R.S.T. system contains a library of primitive operators, (such as MULTIPLY, ADD, SORT, BIT DELAY, ETC.) as well as a number of more complex procedural definitions (such as Biquad, Lattice, Butterfly, etc.) that enable a

range of signal processing architectures. The language compiler produces an intermediate level format file as output. This file is used by both a layout program, which produces the mask geometry, and a simulator. The simulator is event driven, which means that the voltage values on circuit nodes are modeled as discrete bits of data occurring at discrete time intervals. The functioning of individual operators is simulated on a word-by-word basis in response to a file of input commands. It is asserted that the simulator has the ability to uncover timing bugs in the data stream.

A unique and useful aspect of F.I.R.S.T. is incorporation of a translator program to convert the simulator's output into a form suitable for use with an automatic test pattern generator system.

2. Behavioral Specification Compilers

a. Common Properties

In contrast to the floor planners, which accept structural specifications at the top level, are the behavioral specification compilers, which do not require the designer to possess a prior mental model of the architecture to be designed. These systems attempt to translate a high-level behavioral description of the circuit into a geometric mask description. This step is a significant one over floor planners.

b. Ayres' Work

Ayres is the first to have written a book-length treatment of silicon compilation [Ref. 12]. Ayres' compiler approach starts with a synchronous logic specification of the chip behavior. Then follows a decomposition of this specification repeatedly into a hierarchy of implementing

NMOS PLA's which become successively more area-efficient as they become smaller. The system includes heuristics to manage and optimize on-chip routing among the PLA's generated. Ayres' compiler is potentially applicable to a broader class of circuits than F.I.R.S.T., but is still not efficient for a general range of problems. The scope of applications was restricted intentionally to control complexity. The very use of PLA's as the sole basic building block restricts the area efficiency of this system. Even though the PLA's themselves become more area-efficient as they become smaller, the difficulty of managing their interconnections limits the ultimate efficiency of the layout.

c. MacPitts

MacPitts is the only broad spectrum silicon compiler with which this author has had any first-hand experience. It is also the most widely known and most ambitious behavioral specification compiler in operation.

The hardware specification generated by MacPitts is in the form of an NMOS technology CIF file. To cope with the complexity of this project the designers restricted the target architectures to microprocessors consisting of a data path and a controller (see figure 2.3.) Other restrictions include fixing the width of the data path to one value throughout the design, and requiring the designer to specify control and parallelism explicitly. The latter is not actually a restriction in one sense, however, because it affords greater generality in designs. Except for making pin assignments, the MacPitts user has no explicit control over the floor plan of his design. The MacPitts target architecture results in the same basic floor plan for all designs, although this particular architecture is applicable to a greater variety of digital problems than any other scheme presently available.

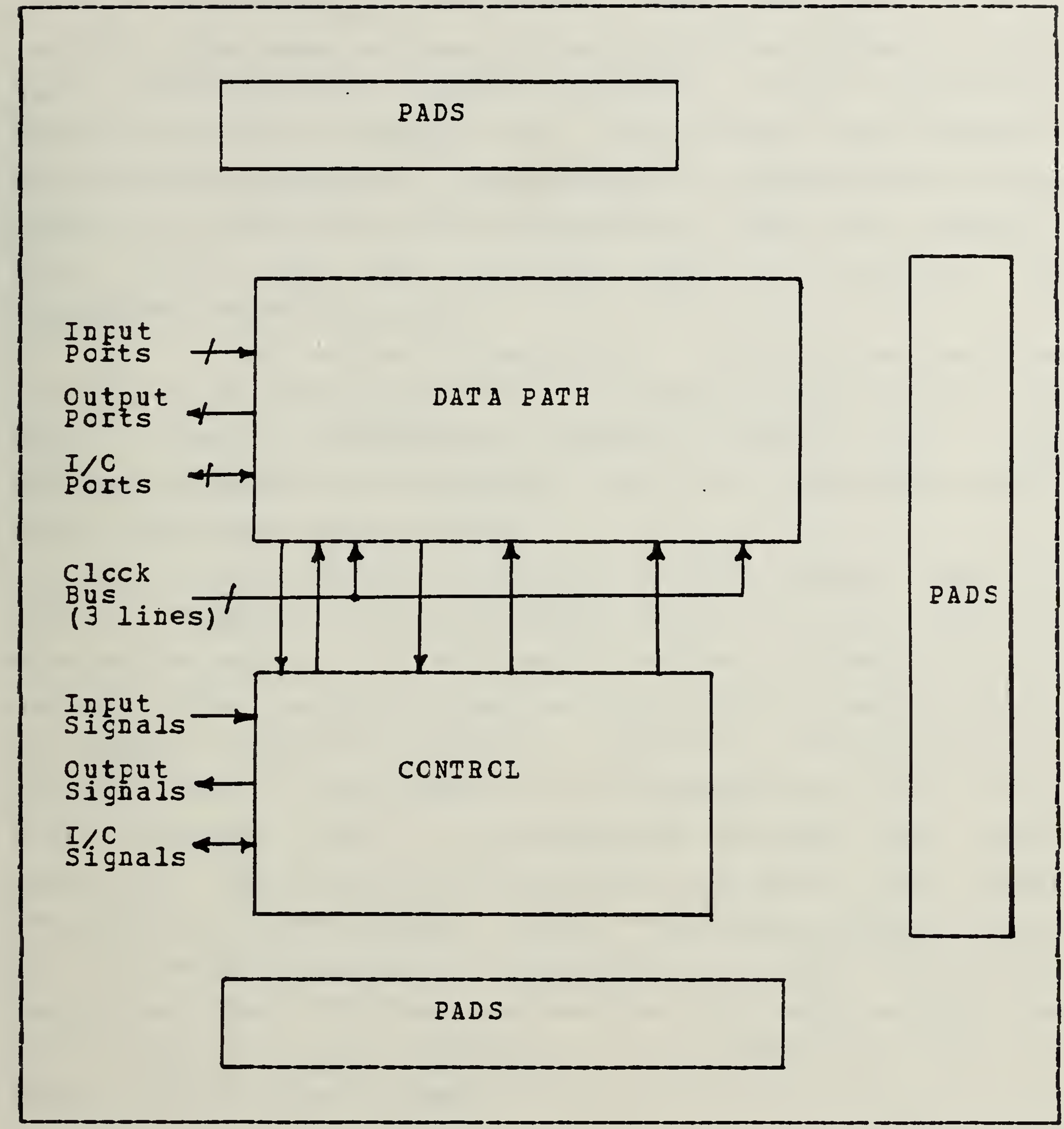


Figure 2.3 Floor Plan of the MacPitts Target Architecture.

The data path portion of the layout consists of a rectangular array of units called "organelles." An organelle is a bit-wise functional unit. A standard library of functions--adder, subtracter, shifters, incrementers, comparators, etc.--is provided. Also, if the algorithmic

behavior specification calls for conditional data flow or looping, the data path may also include multiplexers which have connections for control signals. This multiplexer organelle is not a library cell but is built into MacPitts. Data storage registers, implemented as master-slave flip-flops, are also "built-in organelles." These are instantiated in the data path if their use is implied by the algorithmic specification.

The vertical dimension of the data path outline in figure 2.3 corresponds to the number of bits in the data word. Longer word-lengths produce a taller chip. The various organelles are cascaded along the horizontal dimension of the data path outline.

The control portion of the layout acts on various signals, either derived from the data path or outside the chip, and implements whatever boolean logic is necessary (as inferred from the algorithmic specification) to generate controls signals to drive the multiplexers in the data path. The result is an implementation of a finite state machine, (FSM) as described in Mead and Conway [Ref. 2]. The control unit does not use PLA's, but rather structural NOR gate arrays called "Weinberger Arrays" which can implement arbitrary combinational logic functions. Weinberger [Ref. 15] demonstrates that his logic arrays have three features which contribute to efficiency in an automated circuit layout scheme.

- They simplify the formation of interconnection patterns within the framework of a standardized layout.
- They significantly reduce the required area (by eliminating unused inputs and separate interconnection areas.)
- They eliminate crossing of signal nets (by using single level wiring.)

State timing is controlled not by a two-phase non-overlapping clock, which is somewhat standard in NMOS VLSI, but by a three-phase clock which drives the register circuit shown in figure 2.4. This clocking scheme apparently allows a more compact layout of the register organelle, but requires an extra pin in the package.

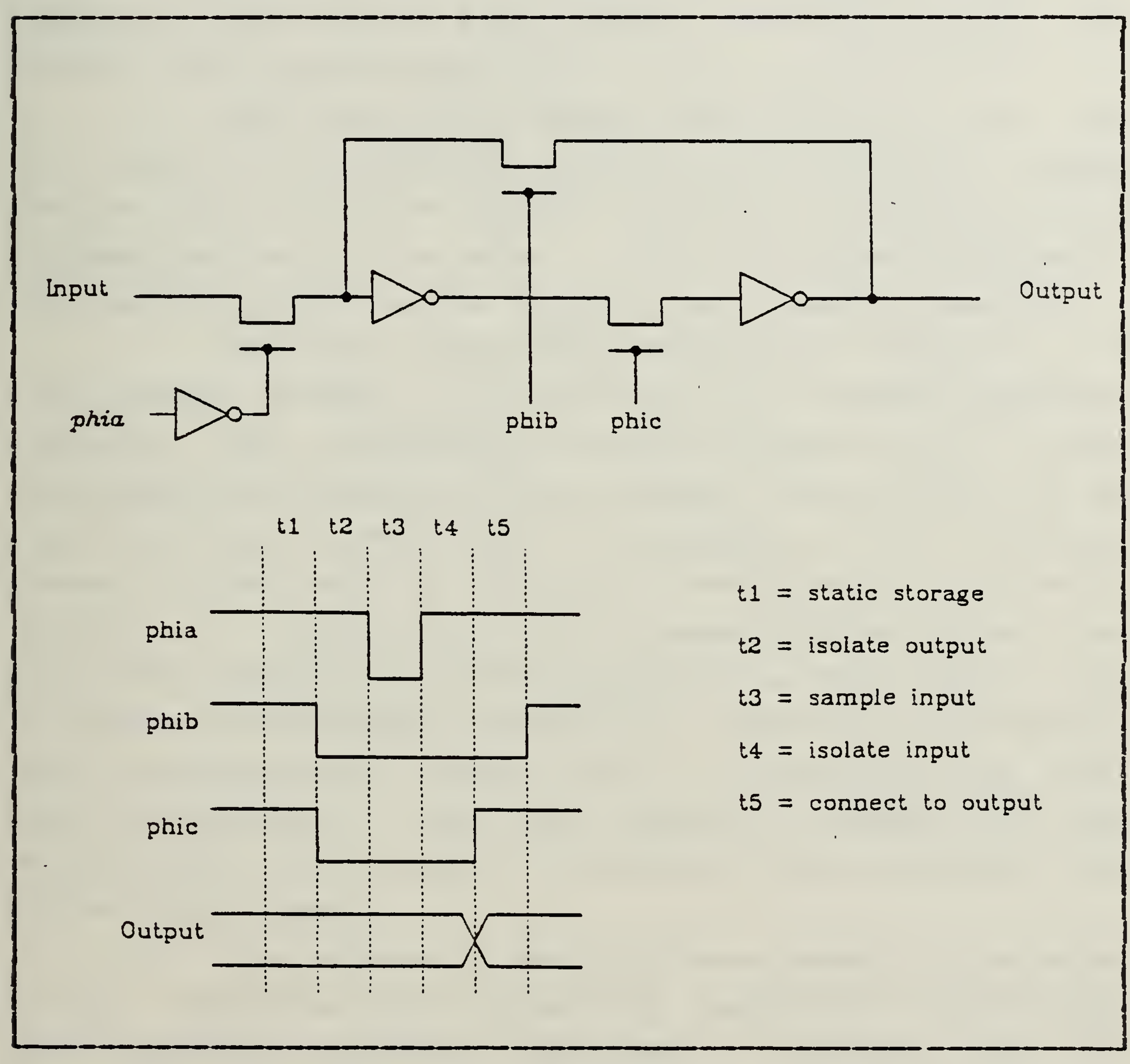


Figure 2.4 MacPitts Register Circuit and Timing Diagram.

One of the authors of MacPitts, Siskind quoted in [Ref. 7], admits that optimizing chip performance was not a primary design goal. Circuit densities reported were 80-100 transistors per square millimeter in 5 micron feature size NMOS--approximately 2 orders of magnitude lower than the state of the art layouts reported in Gross [Ref. 8]. Southard contends that the cells he helped design for MacPitts could fairly easily have been made 20 per cent smaller than they are [Ref. 5].

MacPitts only produces NMOS output in CIF, but the user has a choice of either 4 or 5 micron minimum feature size, which the compiler handles by linearly scaling all features except the pads. The latter are contained in two separate libraries for 4 micron and 5 micron designs.

From the programming viewpoint, MacPitts is a very complex system. It consists of a binary executable module of over 1.5 megabytes which was built up as a LISP programming environment and then dumped, as described in the Franz Lisp manual [Ref. 13]. A synopsis of the functional elements which make up this LISP environment is shown in figure 2.5 . Unlike F.I.R.S.T., these programs (except the functional simulator or "interpreter" as its authors call it) are not individually accessible. MacPitts runs automatically from beginning to end with no possibility for operator intervention. The only control available at the console when the compiler is running is the standard UNIX system abort signal.

The authors of MacPitts were careful to separate all the processing into technology independent (front-end) and technology dependent (back-end) portions, with the intermediate-level description being the point of division. This intermediate-level description is available to the user as an "object file" in human readable form. It is possible, although not very practical, to write an object file

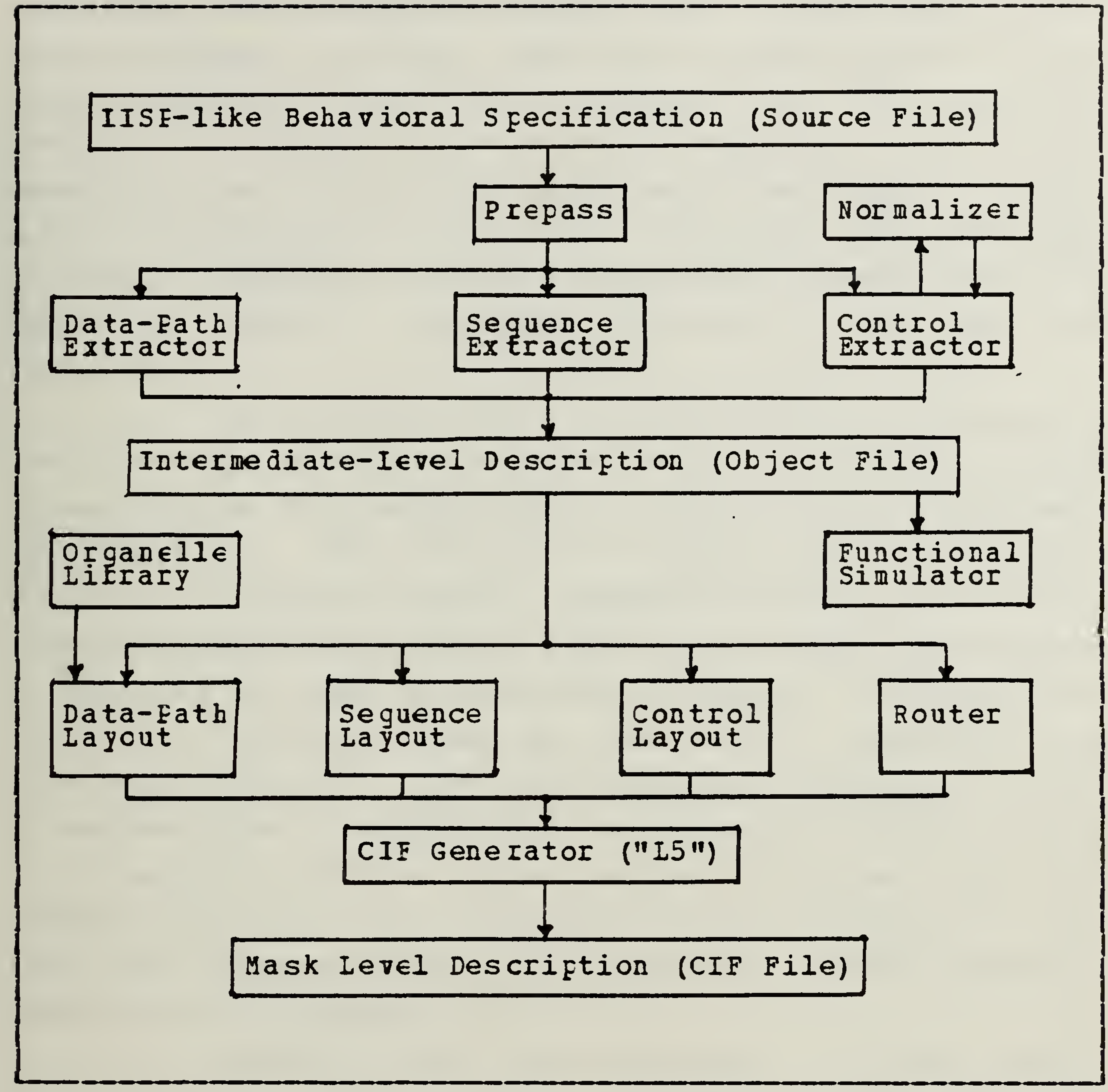


Figure 2.5 MacPitts Program Data Flow.

directly for input to the back end of MacPitts. The object file is a long list containing 5 elements, each element being itself a list. The 5 elements are: definitions, flags, data path, control, and pins. This list is, of course, in a form readable by the layout programs.

The layout programs produce only NMOS technology. As mentioned above, two bonding pad libraries are

included: the Stanford standard cell library pads for 5 micron designs, and the MOSIS ARPA community pads for 4 micron designs. The "layout language" and CIF generation program, L5, which is embedded in MacPitts, was written especially for the project by Crouch [Ref. 14]. It has built-in facilities to handle both NMOS and CMOS technology layouts. Therefore, expanding MacPitts to produce CMOS CIF would not entail a <u>complete</u> rewrite of the back end programs.

An important feature of the MacPitts software is the functional simulator or interpreter. A MacPitts program is not only an IC specification, it is also an algorithmic specification. The interpreter executes the specification program as a general-purpose computer using an interactive, screen-oriented input/output style. By invoking this option of MacPitts the user can exercise his design, thereby validating (to whatever extent the exercise is complete) its functional fidelity. Once the functional simulation is done to satisfaction, MacPitts can be restarted without setting the interpreter option. This produces a finished layout and corresponding CIF file. By using the same language to drive both the interpreter and the integrated circuit compiler, human error is reduced.

MacPitts lacks some features. It has none of the capabilities of F.I.R.S.T. to produce a test pattern to exercise the chip. It also lacks any built-in mechanism to identify worst-case path delays or to predict the maximum clock frequency of the finished chip. It does keep account of conductivity information, however, which it uses to predict chip power consumption.

MacPitts uses a "correct by construction" doctrine in the layout process. By denying the user the means to specify the layout details of the chip, this approach also denies the user the opportunity to commit

design rule errors or to translate the specification program into a non-corresponding layout. But can MacPitts itself make design rule errors?

The following chapters examine how to use MacPitts to produce an integrated circuit layout, how to validate the design, and where to look for ways to improve chip performance.

III. USING MACPITTS

A. THE INPUT FILE

1. Fundamentals of the MacPitts Language

"MacPitts," the system for generating a custom integrated circuit, is also "MacPitts," the language in which the algorithm is specified. In this section the second meaning is the one implied. All of the information which specifies what functional behavior is required of a VLSI circuit is communicated to MacPitts in a single text file. This file, which must have the extension ".mac", is written using syntax which closely resembles that of the LISP programming language. Because the MacPitts compiler is implemented in LISP, it is reasonable to expect the syntax of the MacPitts design language to follow the LISP parenthesized notation. This choice was made by the authors of MacPitts because it eliminates the need for a separate parser.

LISP is a list processing language. Its data elements are "symbolic expressions" made up of "atoms" (fundamental word-like objects separated by spaces), lists of atoms, lists of lists of atoms and so on. One of the strengths of LISP is the ability to concatenate atoms or lists into new lists, and to perform other operations on a list or a hierarchy of lists to produce new lists modified in useful ways. LISP has many built in functional definitions which are an "environment" of specifications for the operations to be performed on lists. These definitions are all contained in The Franz Lisp Manual [Ref. 13]. In addition to using these definitions, the LISP user is free to extend the LISP environment by defining new functions which

specify other operations on lists. The types of operations may be simple manipulations of the atoms by partitioning or permutation, or, if the atoms which comprise the list happen to be numbers, arithmetic operations may be performed. The definitions of the operations themselves may also be assembled from lists of more primitive operational atoms. This functional extension of operations is what the authors of MacPitts have done in creating the MacPitts Lisp environment.

The design of a VLSI circuit can be thought of as a list-building process in which the lists are electrical ports, registers, interconnection nets, data testing operations, and ultimately a string of words which define a unique patterning of silicon in the mask level descriptive language, CIF. These lists are built according to rules contained in another list--the algorithmic specification source file. Although the MacPitts design language resembles LISP syntactically, its semantics is different and much more limited. A powerful feature of LISP is, for example, recursive definition. This feature is absent in the MacPitts design language. A description of the MacPitts grammar in Backus normal form is given in [Ref. 4].

In its most general form, a MacPitts "program" to specify a circuit's behavior consists of a set of "processes," each of which executes sequentially, but all of which run in parallel. The states of each process are fundamentally disjoint from those of the other processes. This allows the hardware for each process to run independently of the other processes, if desired, and concurrently with the states of the other processes, in any case. The operations performed by a given process in a given state are specified by a "form." Each form corresponds to a single machine state, and is executed in one clock cycle. A state may be given a name by preceding the form with a label.

Normally execution proceeds sequentially from one state to the following state in the .mac file at each clock cycle. A "go" form can be used, however, to deviate from this sequential flow by causing the named state to be executed next instead of the syntactically following state.

Data is communicated between the data path and the external world through "ports" which have the same bit width as the data path. Only a single data path width definition is allowed per program. A port may be declared "input," "output," "tri-state output," or "i/o." Ports may also be declared as "internal," in which case they simply cascade the output of one data path operation to the input of another. The data path may also be specified to contain registers. The difference between internal ports and registers is that registers can store data indefinitely after it has been clocked in, whereas ports are only electrical nodes in the data path and therefore do not store data. Ports simply are arrays of named terminals for conducting data from one point to another.

Control of operations performed on the data by the data path organelles is governed by the Weinberger array control unit. Control outputs from the control unit to the data path may determine, by means of their control over multiplexer organelles within the data path, which operations occurring within the data path will affect downstream organelles. Status outputs from the data path returning to the control unit allow the sequence of operations performed by the control unit to vary depending on the data present either in the registers or at any other point in the data path. The control unit functions may also be made to depend upon external inputs. The control unit communicates with the outside world using "signals," which are analogous to the "ports" used by the data path except that each signal appears on a single wire. Signals may be declared as "input," "output," "tri-state output," "i/o" or "internal."

Operations performed by the data path during a given state are specified by the LISP "setq" form. The setq causes the data path to evaluate a sequence of operations on either input port data, internal port data or register data. (The setq may also be used with signals.) The result of these specified operations is then conducted to another named port or loaded into a data path register during the next clock cycle. The compiler includes enough copies of each operator in the data path so that separate processes, intended to run in parallel, do not conflict over the attempted shared use of a single resource. The data path can cascade several operations together in a single form. This allows forms such as the following example, which computes a=b-c using 2's complement arithmetic, to execute in one clock cycle:

```
(setq a (+ b (1+ (not c)))) .
```

The list consisting of everything on the preceding line is a single form. There are three operators in this expression: "+," which specifies use of an adder, "1+" which specifies an incrementer, and "not" which specifies an inverter. Each operator is followed by its operands listed in <u>symbolic</u> notation. Therefore, the single operand of 1+ is the integer that results from <u>evaluating</u> the expression "(not c)." Note that there is not a default hierarchy of operations within a form. As with LISP, the order of operations in MacPitts must be specified explicitly by the use of nested parentheses.

Sequences of setq forms normally operate sequentially, each being executed on a separate clock cycle. By enclosing the forms within another "parallelizing form," of which "par" is an example, several forms can be made to run in parallel, gaining speed over sequential operation at the cost of more hardware and hence more area in silicon. The par form is used as follows:

```
              (par form1 form2 form3...)   .
```

Of course the results obtained by running setq forms in parallel may be quite different from those obtained by running them all sequentially within one process. Consider the following example where "a" and "b" have already been declared registers (i.e. master-slave flip flops):

```
             (par (setq a b)
                  (setq b a)) .
```

This expression will result in exchanging the contents of "a" with contents of "b." The exchange will be done in one MacPitts clock cycle. This action is made possible by the input isolation which occurs during the flip-flop operating cycle. All such data storage elements are read before they are written. On the other hand, sequential operation of the same setq's is implied in the following process:

```
             (process load1 (setq a b)
                            (setq b a)) .
```

This process will load both b and a with the original contents of b, and require two cycles to do it. (Here "load1" merely furnishes a process name, as demanded by the MacPitts grammar.) We have used two lines and indented format only for the sake of clarity. All the functional information needed by MacPitts is denoted by the ordering of forms within the nests of parentheses.

The "cond" form allows the conditional execution of other forms it contains during a given state. It consists of a list of guards, only one of which is to be executed. Each guard begins with a "condition" which determines whether the remaining forms in the guard are to be executed. The first guard whose condition is true enables the execution of the forms following the condition in that guard. This is illustrated by the following example adapted from [Ref. 4].

```
(cond (condition1 (cond (condition2 form1 form2)
                        (condition3 form3 form4 form5)
                        (t form6)))
      (condition4 (cond (condition5 form7 form8))
                  (cond (condition6 form9))
                  form10))  .
```

This example is heavily nested. Nevertheless, close examination reveals that the outermost "(cond..." has only two guards in its list, each of which contains other "(cond..." forms. The two guards are:

```
      (condition1 (cond (condition2 form1 form2)
                        (condition3 form3 form4 form5)
                        (t form6)))
```

and

```
      (condition4 (cond (condition5 form7 form8))
                  (cond (condition6 form9))
                  form10)  .
```

If condition1 is false and condition4 is true then form10 is executed. If condition5 is true then form7 and form8 are executed along with form10. Likewise if condition6 is true then form9 is executed in parallel as well.

The semantics of the cond statement is inherently parallel. The conditions of the alternate guards are checked in parallel. Likewise, all forms within the guards are executed simultaneously in one clock cycle. The compiler makes the conditions of different guards in one cond form mutually exclusive, and implements them using combinational logic in the control unit as described above. This logic is used to enable or inhibit the execution of forms controlled by that guard in parallel.

Note that the form:

```
(cond (t form1 form2 form3 ...))
```

is used to enable parallel execution of several forms during one clock cycle without being dependent on <u>any</u> condition. (The "t" stands for "true.") The "(par..." form already encountered is actually just a shorthand macro expression for the "(cond (t..." form.

In a MacPitts layout, the conditions are formed in the control unit, which is a Weinberger array of NOR gates [Ref. 15]. Therefore, they are not limited to only the sum-of-products notation used by PLA-based finite state machine compilers. The conditions are derived from either signals arriving on an input pin, signals from the data path, or signals arriving from other processes. More complex conditions can be constructed from these signals using the logical operators "and," "or" and "not" to build arbitrary Boolean expressions. These operators are part of the MacPitts library of functions. Thus, the cond statement is one of the most powerful features for providing high performance designs.

With this brief and somewhat condensed description of the features available in the MacPitts algorithmic language, the way is prepared to to understand an example of some code which will produce a complete integrated circuit chip. A full detailed description of all the facilities of MacPitts is found in a report authored by its creators [Ref. 16], which also serves as a fairly complete users' manual.

2. <u>Two Multiplier Examples</u>

Consider, line by line, figure 3.1 which is a listing of the file multic.mac. This example and the one which follows it are inspired by similar ones in [Ref. 16]. It contains all of the design information needed by MacPitts to produce a 4 bit combinational multiplier. On any line,

```
 1  ; multiplier, no state combinational
 2  (program multic 4
 3     (def 1 ground)
 4     (def ain port input (2 3 4 5))
 5     (def bin port input (6 7 8 9))
 6     (def res port output (10 11 12 13)); result
 7     (def r0 port internal)
 8     (def r1 port internal)
 9     (def r2 port internal)
10     (def 14 phia)
11     (def 15 phib)
12     (def 16 phic)
13     (def 17 power)
14     (always
15      (cond ((bit 0 bin) (setq r0 (>> (bit 0 ain) ain)))
16            (t (setq r0 0)))
17      (cond ((bit 1 bin) (setq r1 (>> (bit 0 (+ r0 ain)) (+ r0 ain))))
18            (t (setq r1 (>> (bit 0 r0) r0))))
19      (cond ((bit 2 bin) (setq r2 (>> (bit 0 (+ r1 ain)) (+ r1 ain))))
20            (t (setq r2 (>> (bit 0 r1) r1))))
21      (cond ((bit 3 bin) (setq res (>> (bit 0 (+ r2 ain)) (+ r2 ain))))
22            (t (setq res (>> (bit 0 r2) r2))))))
```

Figure 3.1 Multic.mac Source File.

text following a semicolon is treated as a comment, which the compiler ignores. Line 2 tells the compiler that a "program" (which is another way of saying, "circuit design") called "multic" starts here, and that the data path is 4 bits wide. Because the data path is only 4 bits, this simple multiplier will only be able to output numbers from 0 to 15. Even though the input ports are also four bits wide, we must restrict input numbers to only those whose product falls in the range of values from 0 to 15. Furthermore, if this algorithm is to give correct results for all multipliers, without overflow, the leading bit of the multiplicand must be zero. No provision is made to output a flag if the dynamic range of the multiplier is exceeded.

Lines 3 through 13 declare the various signals and integer data words input to, output from and existing within the multiplier. Line 3 assigns the ground connection to pin 1 which is always in the upper left corner of the layout;

subsequent pin numbers proceed clockwise from this point around the layout perimeter. Line 4 assigns pins 2-5 to an input port labeled "ain." This input is the multiplicand. By MacPitts convention, the most significant bit (MSB) of ain is read from the first pin on the list, pin 2, and the least significant bit (LSB) from the last pin on the list, pin 5. Line 5 similarly defines the multiplier input port, "bin." Line 6 assigns an output port labeled "res" (for result) to another block of 4 pins. This port also serves as the accumulator for the fourth and final partial product. Lines 7 through 9 define 3 internal ports (necessarily of width 4 bits) labeled r0, r1 and r2. These serve to cascade the three stages of a standard shift and add algorithm. Each port contains one of the first three partial products, each being the result of operations conditioned on one of the multipler bits. Lines 10 through 12 assign pins to the three phase clock, whether that clock is used by the circuit or not. In multic.mac the clock is not used. Line 13 defines the + 5 volt direct current power, Vdd, connected to pin 17.

Line 14 signifies that the functions which follow, up to the matching right parenthesis on line 22, are to execute on every clock cycle. The "(always..." form is really the "(process..." form, reduced to a single state. Moreover in this case, given the (always... form, and given that the data path contains only ports and not registers, the inputs will affect the result after an interval governed only by the sum of the physical gate delays in the data path and control unit. There is no controlled latency in the data path, because there are no registers in this design in which to store data.

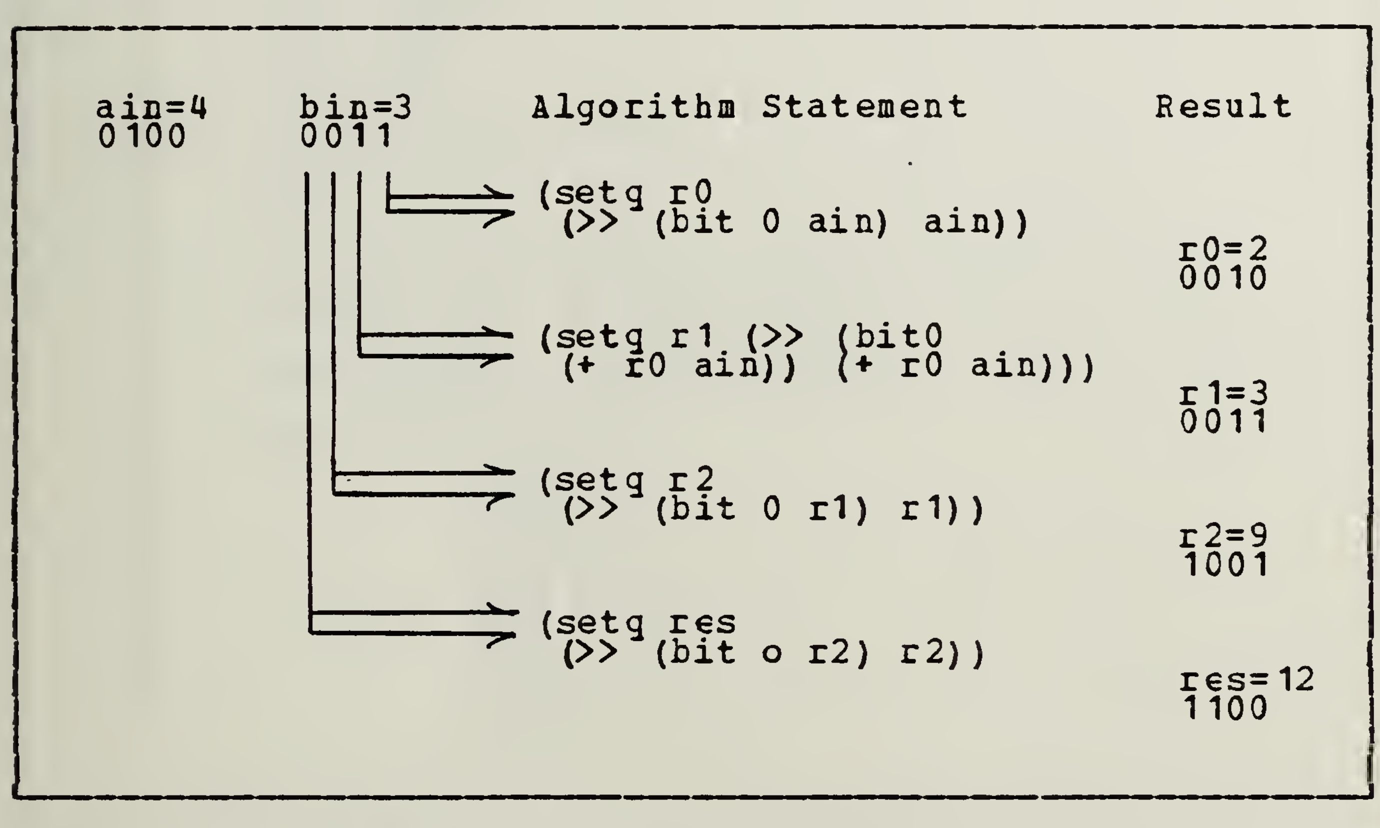
Lines 15 through 23 contain the shift and add scheme. In lines 15 and 16 the controller is told to examine bit 0 (the LSB) of bin. If it is high (true) the r0

port takes on the value of the ain port rotated right by one bit, i.e. r0 is actually connected by means of a multiplexer organelle to a right rotated version of ain. The shift-right-one-bit form, ">>," takes two arguments. The second argument specifies what data word is being shifted, and the first tells what to put in the MSB of that data word. Thus, a rotate is also within the capabilities of the shift form, as it is applied in this case. If bit 0 of bin is not high, then, by line 16, the r0 port--all 4 bits--is connected to ground. In lines 17 and 18 the controller is told to examine bit 1 of bin. If it is high, then r1, the next internal port in the data path, is connected to a right-rotated version of the sum of r0 and ain. The adder organelle in MacPitts performs this summation as a standard ripple carry full addition. Note again that the expression:

```
(bit 0 (+ r0 ain))
```

in line 17 turns the single shift operator into a right rotate operator by making the MSB of r1 contain the same value as bit 0 of the sum of r0 and ain. If bit 1 of bin is low, on the other hand, line 18 instructs the controller to connect r1 to simply a right-rotated version of r0. Note that no rotations are being performed by any of these operations in the sense that a shift register would perform them. It is only the interconnections between organalles that are being set up variously by the controller to give an appearance of forwarding a rotated version down the data path. Also note that even though the addition form appears twice in line 17, logically only one adder need be instantiated, since the operands are identical in both occurrences. MacPitts, too, can recognize this, and will not waste space creating more adders than the minimum necessary. In lines 19 and 20 the controller examines bit 2 of bin. If it is high, port r2 is connected to a right-rotated version of the sum of r1 and ain. If bit 2 of bin is low, r2 is connected

to a right-rotated version of r1. In lines 21 and 22 the controller finally examines the MSB, bit 3, of bin. If it is high, the output port, res, is connected to a right-rotated version of the sum of r2 and ain. If bit 3 of bin is low, res is connected to a right rotated version of r2. For concreteness, a schematic trace of this algorithm in action on the problem "4x3=12" is presented in figure 3.2.



```
(setq r0
 (>> (bit 0 ain) ain))

(setq r1 (>> (bit0
 (+ r0 ain)) (+ r0 ain)))

(setq r2
 (>> (bit 0 r1) r1))

(setq res
 (>> (bit o r2) r2))
```

Figure 3.2 Example of the Multic Behavioral Specification.

For comparison, consider now another design. This one is specified by the file multip.mac shown in figure 3.3 This is a four bit pipelined multiplier in which the product does not appear at the result port until the third clock cycle after values have been applied to the inputs, ain and bin. Changing the combinational design to a pipelined design can most easily be accomplished in two steps. First,

[illegible]

[illegible]

[illegible]

```
; multiplier, with pipelining
(program multip 4
   (def 1 ground)
   (def ain port input (2 3 4 5))
   (def a0 register)
   (def a1 register)
   (def a2 register)
   (def bin port input (6 7 8 9))
   (def b0 register)
   (def b1 register)
   (def b2 register)
   (def res port output (10 11 12 13))
   (def r0 register)
   (def r1 register)
   (def r2 register)
   (def 14 phia)
   (def 15 phib)
   (def 16 phic)
   (def reset signal input 17)
   (def 18 power)
   (always
   (cond ((bit 0 bin) (setq r0 (>> (bit 0 ain) ain)))
         (t (setq r0 0)))
   (cond ((bit 1 b0) (setq r1 (>> (bit 0 (+ r0 a0)) (+ r0 a0))))
         (t (setq r1 (>> (bit 0 r0) r0))))
   (cond ((bit 2 b1) (setq r2 (>> (bit 0 (+ r1 a1)) (+ r1 a1))))
         (t (setq r2 (>> (bit 0 r1) r1))))
   (cond ((bit 3 b2) (setq res (>> (bit 0 (+ r2 a2)) (+ r2 a2))))
         (t (setq res (>> (bit 0 r2) r2))))
   (cond (reset (setq a0 0)
                (setq b0 0)
                (setq a1 0)
                (setq b1 0)
                (setq a2 0)
                (setq b2 0))
         (t     (setq a0 ain)
                (setq b0 bin)
                (setq a1 a0)
                (setq b1 b0)
                (setq a2 a1)
                (setq b2 b1)))))
```

Figure 3.3 Multip.mac Source file.

the three internal ports of multic, r0, r1 and r2, are all redefined as registers. Then six other new registers, a0-a2 and b0-b2 are defined to send successive values of the inputs ain and bin down the pipe in step with their corresponding partial products. The ease with which this is done (from a user's point of view) is evidence of the power of MacPitts to create custom designs.

Referring to figure 3.3 we see that the shift and add algorithm, lines 22-29, is identical to that of multic.mac. In line 19 pin 17 is defined as a "reset" signal input. The reset signal is <u>required</u> for any MacPitts design which uses one or more "process" forms in order that the program counters for all processes can always be reset to the same known state. This is obviously vital when two or more processes on the same chip must be synchronized. In the multip design, however, which uses the "(always..." form, the reset signal performs no such built in automatic function. The reset signal is available, however, for user-specified functions as well, and in this case is used only to signal a setq of all internal multiplier and multiplicand registers to zero, instead of passing the values one more step down the pipeline. Therefore, the reset is not essential to the pipeline multiplier operation here but only acts to allow the pipeline to be emptied out and to inhibit any new input data from propagating to completion, for what that may be worth in whatever the intended application. It is included here for illustration only. Recall that propagation of all input data in the pipeline (lines 30-35 or, if reset is false, lines 36-41) occurs in a single clock cycle as well, because these setq's are enclosed in the "(cond..." form, which causes them to be executed in parallel.

B. INVOCATION OPTIONS

Equipped with one or more .mac files written to reflect the desired behavior of a circuit, the user is ready to run macpitts.[1] The form of the command line invocation from the UNIX shell is simply

```
% macpitts <program_name> <options>
```

---

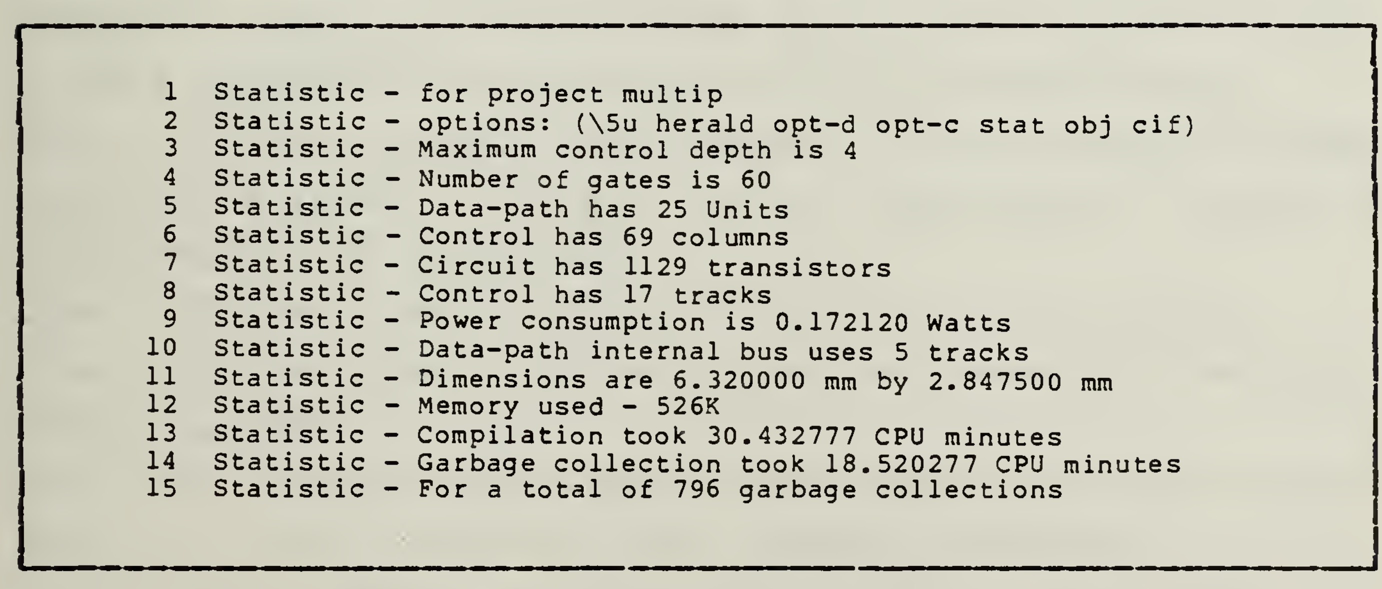
[1] The name assigned to the executable binary file on the UNIX operating system which embodies the MacPitts system is "macpitts."

where <program_name> would be either multic or multip, in the case of the previous examples, and <options> is any or none of the words from the list:

stat*	nostat*
herald	noherald*
cif*	nocif
obj*	noobj
int	noint*
opt-d*	noopt-d
opt-c*	noopt-c
4u	5u*

where the * options are the defaults and the left and right columns are mutually exclusive.

The "stat" option tells macpitts to output statistics about the chip design to the standard output device (terminal screen, normally) as various parameters are calculated. Figure 3.4 shows the statistics generated for the multip chip. The meaning of these statistics is as follows.



```
 1  Statistic - for project multip
 2  Statistic - options: (\5u herald opt-d opt-c stat obj cif)
 3  Statistic - Maximum control depth is 4
 4  Statistic - Number of gates is 60
 5  Statistic - Data-path has 25 Units
 6  Statistic - Control has 69 columns
 7  Statistic - Circuit has 1129 transistors
 8  Statistic - Control has 17 tracks
 9  Statistic - Power consumption is 0.172120 Watts
10  Statistic - Data-path internal bus uses 5 tracks
11  Statistic - Dimensions are 6.320000 mm by 2.847500 mm
12  Statistic - Memory used - 526K
13  Statistic - Compilation took 30.432777 CPU minutes
14  Statistic - Garbage collection took 18.520277 CPU minutes
15  Statistic - For a total of 796 garbage collections
```

Figure 3.4 Compiler Statistics for multip.

Line 1 simply echoes the program name which was given at the beginning of the multip.mac source file.

Line 2 summarizes the invocation options in effect either by user selection or default.

Line 3 gives the worst-case number of logic levels between any input and any output in the control unit.

Line 4 gives the total number of NOR gates needed in the control unit.

Line 5 is the number of data path "organelle units," where an organelle unit is a word-length assembly of organelle bits. This number is the same as the number of elements in the data path list of the multip.obj file.

Line 6 is the number of vertical metal columns in the control array, excluding the ground columns.

Line 7 is the total number of transistors in the circuit, including the data path, control unit, and all bonding pads.

Line 8 is the stack height of horizontally running polysilicon lines used to intraconnect the control unit.

Line 9 is an estimate of the worst-case static power consumption of the chip obtained using the layout topology, heuristic values of undetermined origin for the conductivity of each electrical feature, and a 5 volt power supply.

Line 10 is the maximum stack height of horizontally placed polysilicon lines, per bit in the data path, needed to interconnect the organelles.

Line 11 is the overall outline size of the chip layout.

Line 12 is the peak storage allocation demanded by macpitts during the run.

Line 13 is the CPU time required for compilation and layout, which is always less than the apparent running time by an amount which depends on the average system usage rate.

Lines 14 and 15 reflect a function of Franz Lisp wherein past used storage locations are reclaimed for the available memory list. The last three statistics were probably included because macpitts can be very demanding of computing resources.

The "herald" option outputs messages to the terminal screen at each milestone in the sometimes lengthy compilation process. These reassure the user that macpitts is still running. In addition to heralding what point in the design process macpitts is currently working on, information on current accumulated CPU time and CPU garbage collection time is printed at the beginning of each herald line in units of sixtieths of a second.

The "cif" option keys the compiler to output a mask level description .cif file in the Caltech Intermediate Form. The cif option is normally not deselected unless the available disk storage space is limited and the user is only interested in reading the statistics for his compiled design. (The cif file for a relatively simple design, multip.cif, is over 158 kilobytes long.) If no cif is produced on a given macpitts run, the entire layout process must be repeated to subsequently obtain a cif file. This is done most expeditiously by running macpitts with the noobj option.

The "noobj" option tells macpitts to start with a previously created object file (the output of the macpitts "front end,") rather than a source file. MacPitts will then effectively start at the "back end," doing the layout and outputing statistics and cif, assuming these are included in the options list.

"Int" tells macpitts to use the interpreter mode, which allows functional simulation of the chip without actually performing the layout and generating a .cif file.

"Opt-c" and "opt-d" invoke optimization routines for normalization of the combinatorial logic of the control unit. Investigation of the four possible combinations of these two options reveals that they do not affect the overall dimensions of the final 8 bit multiplier design (to be described later.) This is probably because the pins,

data path layout and bus wiring dominate the chip area, not the control unit, which is comparatively small for this chip. The compilation time required, however, was approximately 20 percent greater when opt-c and opt-d were used than when they were not used. Using opt-c and opt-d does reduce the complexity of the control unit, and therefore will reduce signal delays, to the benefit of operating speed.

The "4u" option sets the minimum feature size for the layout to 4 microns, and accordingly lambda, the commonly used parameter which represents the half line width dimension, is set to 200 centimicrons.

Another option, logo, was available in the original macpitts, but is not supported at NPS because suitable font files are not currently available.

C. USE OF THE MACPITTS INTERPRETER

Invoking macpitts with the int option should be the first step in every Macpitts design cycle. Macpitts has good facilities for catching grammatical errors in the user's .mac source code which operate whether or not the interpreter is invoked. After the .mac file passes grammar checks, the interpreter allows the extracted algorithmic description to be exercised with arbitrary inputs. The results are displayed on the screen to provide an indication that the design is functionally correct. Assuming the user's path list is set up in the .login file to include the directory, /vlsi/macpit, the following command can be issued:

```
% macpitts multip int herald
```

This will cause macpitts to scan the multip.mac source file and extract from it the circuit behavior information. Then macpitts will display a table of all declared ports,

registers, flags, signals and processes, noting that they are all currently undefined. The user may select for display, at this point, a menu of interactive commands which clearly states how to interact with the interpreter. The user can set the values of input ports and signals as desired. Not all internal ports will necessarily be defined simply by setting the input ports. Generally several clock cycles must be simulated before the chip internals are all defined. Macpitts tells the user which antecedants stand in the way of resolving data definitions. Next the user will probably single step (or multi step) the macpitts clock while observing the effect on the internal registers and output port(s) after each cycle. There is also provision to write out the current state of the circuit to a file, multip.int. Any number of states can be saved by appropriate renaming of files as they are written. Since macpitts does not allow the user to specify different file names for each state saved, newly written .int files can immediately be renamed uniquely from an adjacent terminal logged on to the same account as the one running macpitts. This is completely feasible under UNIX.

As an example, figure 3.5 shows a concatenated listing of 4 such files from a single session with the macpitts interpreter. As would be expected, the format of these files is that of a LISP list, whose meaning can be clearly inferred because it follows the same syntax as the MacPitts language itself. The first file, lines 1-14, is a dump of the state of the circuit after setting the input ports ain and bin to 4 and 3, respectively, and the reset signal to false. Note that all data downstream of the inputs is still undefined at this point. Lines 16-28 show the result after one clock cycle. Lines 30-42 show the result after two clock cycles. Lines 44-56 show the result after the third clock cycle when the result, 12, is present for the first

```
% cat -n multip.int[1-4]
     1
     2  ((register a0 undefined-integer)
     3   (register a1 undefined-integer)
     4   (register a2 undefined-integer)
     5   (register b0 undefined-integer)
     6   (register b1 undefined-integer)
     7   (register b2 undefined-integer)
     8   (register r0 undefined-integer)
     9   (register r1 undefined-integer)
    10   (register r2 undefined-integer)
    11   (port ain 4 console)
    12   (port bin 3 console)
    13   (port res undefined-integer chip)
    14   (signal reset f console))
    15
    16  ((register a0 4)
    17   (register a1 undefined-integer)
    18   (register a2 undefined-integer)
    19   (register b0 3)
    20   (register b1 undefined-integer)
    21   (register b2 undefined-integer)
    22   (register r0 2)
    23   (register r1 undefined-integer)
    24   (register r2 undefined-integer)
    25   (port ain 4 console)
    26   (port bin 3 console)
    27   (port res undefined-integer chip)
    28   (signal reset f console))
    29
    30  ((register a0 4)
    31   (register a1 4)
    32   (register a2 undefined-integer)
    33   (register b0 3)
    34   (register b1 3)
    35   (register b2 undefined-integer)
    36   (register r0 2)
    37   (register r1 3)
    38   (register r2 undefined-integer)
    39   (port ain 4 console)
    40   (port bin 3 console)
    41   (port res undefined-integer chip)
    42   (signal reset f console))
    43
    44  ((register a0 4)
    45   (register a1 4)
    46   (register a2 4)
    47   (register b0 3)
    48   (register b1 3)
    49   (register b2 3)
    50   (register r0 2)
    51   (register r1 3)
    52   (register r2 9)
    53   (port ain 4 console)
    54   (port bin 3 console)
    55   (port res 12 chip)
    56   (signal reset f console))
```

Figure 3.5 A MacPitts Interpreter Session for multip.

time on the output port. Note that at this point the input data, which was never changed during this session, has also propagated down the three stage pipeline. Of course, one would normally not use a pipelined processor with static data, because the advantage of higher throughput is wasted. The exercise only serves to demonstrate the behavior of the interpreter option.

Two points of practical interest should be made before closing the interpreter discussion. First, it should be observed that the bottom lines of text in the terminal display will be jumbled on the ADM-36 terminals because the /etc/termcap libraries in UNIX version 4.2 differ slightly from those in UNIX version 4.1. Proper screen presentation is obtained, however, if the GIGI terminal is used. Second, the interpreter runs very slowly. It is not unusual during hours of heavy system useage for one to two minutes of terminal time to elapse while the interpreter is processing a single command to cycle the clock. At night, with only 2 users logged on, this clocking operation only takes ten to fifteen seconds.

D. EVOLUTION OF THE 8 BIT PIPELINED MULTIPLIER

1. Design Motivation and Constraints

One possible application for a digital pipelined multiplier of unsigned integers is as part of a high speed digital filter realization. Work done by Loomis and Sinha [Ref. 17] indicates that the impact of pipelining delays on the behavior of digital recursive filters can be compensated for by adjusting the filter weights. Furthermore, their work shows that the stability of the filter can be improved by increasing the number of pipeline stages. It was decided that the design of a multiplier for such applications could be a suitable vehicle from which to study the MacPitts compiler.

The design of circuits which can be fabricated using the available ARPA/MOSIS implementation service is constrained by two standard parameters: a maximum project size of 6890 x 6300 microns, and a maximum bonding pad count of 64 pins. To fully explore the capabilities of MacPitts, it is probably most enlightening to proceed in steps toward the ultimate design.

2. First Design: 3 Stages, 8 Bits on One Chip

To better appreciate the issue involved, the first design is an expansion of multip.mac to an 8 bit wide data path with enough "cond" forms to realize an 8 bit multiplication. Note, however, that the MSB of the multiplicand (ain) must be zero to avoid overflows of the partial product and results ports. Two output ports are used, one for the high order 8 bits of the result (hres), and one for the low order 8 bits of the result (lres). Together these ports form a 16 bit product. One expects the hres MSB always to be zero because the largest valid product is 127x255=32385, which is less than 2^{15}. Because the design has three sets of registers, there are three stages of pipelining, and there is room in the chip for three distinct multiplication problems to be in process simultaneously. A speed vs. area tradeoff is effected by alternating ports with registers in the data path. Ports consume less area than registers. However, ports also introduce more delay in the pipeline stages (whose boundaries are defined by registers) thereby lowering the maximum clock frequency. To further save area, the multiplier bits from bin share space in the low order intermediate results registers (lr0, lr1, lr2) and ports (lp0, lp1, lp2, lp3, lres) by using the following device: after each bit of the multiplier is tested, it is shifted off the right end of the register/port, leaving room at the left end for another bit of the low order result to be

shifted in. The source file for this design, multip8.mac, is shown in figures 3.6 and 3.7. This file was arrived at after first considering what resources would be needed to perform the multiplication. Then register/port templates were written down on paper, and the flow of data traced for a specific case. Next the algorithm depicted by the data flow was translated into MacPitts language resulting in a diagram resembling the style of figure 3.2. Finally the definitions, conditions, and reset functions were added to complete the multip8.mac file. Figure 3.8 partially illustrates the manner in which this was done for the example 104x22=2288. Only the first pipeline stage is shown, representing the first two multiplier bits.

Figure 3.9 shows the linear arrangement of the ports and registers in the data path for this multiplier, as well as the placement of shift and add organelles. The flow of data is down the page. The large size of the full adders relative to the other organelles is not reflected in the scale of this figure. The resulting macpitts layout for this design measures 11848 x 4897.5 microns, which is far too large to be fabricated in a standard MOSIS run. It appears that the design must therefore be partitioned in some way among two or more chips. Ideally, these partitioned "partial multipliers" should be identical in design if fabrication and testing costs are to be minimized.

3. First Partitioning: 2 Bits, 1 Stage Pipeline

The multip8 design may be partitioned in a number of ways. The first approach might be to process two multiplier bits on the chip using one register stage and then one port stage to hold the two partial products in a single pipeline stage, then pipe the partial result to another identical chip. Such a design requires 4 chips to do a complete 7 bit by 8 bit multiplication with 4 stages of pipelining in all.

```
; 3-stage pipelined multiplier, product is 16 bit unsigned integer
(program multip8 8      ; data path is 8 bits wide
  (def 1 ground)
  (def ain port input (2 3 4 5 6 7 8 9))        ;multiplicand
  (def bin port input (10 11 12 13 14 15 16 17))        ;multiplier
  (def a0 register)
  (def a1 register)
  (def a2 register)
  (def hp0 port internal)
  (def lp0 port internal)
  (def hr0 register)
  (def lr0 register)
  (def hp1 port internal)
  (def lp1 port internal)
  (def hr1 register)
  (def lr1 register)
  (def hp2 port internal)
  (def lp2 port internal)
  (def hr2 register)
  (def lr2 register)
  (def hp3 port internal)
  (def lp3 port internal)
  (def hres port output (18 19 20 21 22 23 24 25)) ;high bits of result
  (def lres port output (26 27 28 29 30 31 32 33)) ;low bits of result
  (def 34 phia)
  (def 35 phib)
  (def 36 phic)
  (def reset signal input 37)
  (def 38 power)
; end of definitions
  (always
  (cond ((bit 0 bin)
           (setq hp0 (>> ain))
           (setq lp0 (>> (bit 0 ain) bin)))
         (t
           (setq hp0 0)
           (setq lp0 (>> bin))))
  (cond ((bit 0 lp0)
           (setq hr0 (>> (+ hp0 ain)))
           (setq lr0 (>> (bit 0 (+ hp0 ain)) lp0)))
         (t
           (setq hr0 (>> hp0))
           (setq lr0 (>> (bit 0 hp0) lp0))))
  (cond ((bit 0 lr0)
           (setq hp1 (>> (+ hr0 a0)))
           (setq lp1 (>> (bit 0 (+ hr0 a0)) lr0)))
         (t
           (setq hp1 (>> hr0))
           (setq lp1 (>> (bit 0 hr0) lr0))))
  (cond ((bit 0 lp1)
           (setq hr1 (>> (+ hp1 a0)))
           (setq lr1 (>> (bit 0 (+ hp1 a0)) lp1)))
         (t
           (setq hr1 (>> hp1))
           (setq lr1 (>> (bit 0 hp1) lp1))))
  (cond ((bit 0 lr1)
           (setq hp2 (>> (+ hr1 a1)))
           (setq lp2 (>> (bit 0 (+ hr1 a1)) lr1)))
         (t
           (setq hp2 (>> hr1))
           (setq lp2 (>> (bit 0 hr1) lr1))))
```

Figure 3.6 Multip8.mac Source File.

```
(cond ((bit 0 lp2)
         (setq hr2 (>> (+ hp2 a1)))
         (setq lr2 (>> (bit 0 (+ hp2 a1)) lp2)))
       (t
         (setq hr2 (>> hp2))
         (setq lr2 (>> (bit 0 hp2) lp2))))
(cond ((bit 0 lr2)
         (setq hp3 (>> (+ hr2 a2)))
         (setq lp3 (>> (bit 0 (+ hr2 a2)) lr2)))
       (t
         (setq hp3 (>> hr2))
         (setq lp3 (>> (bit 0 hr2) lr2))))
(cond ((bit 0 lp3)
         (setq hres (>> (+ hp3 a2)))
         (setq lres (>> (bit 0 (+ hp3 a2)) lp3)))
       (t
         (setq hres (>> hp3))
         (setq lres (>> (bit 0 hp3) lp3))))
(cond (reset
         (setq a0 0)
         (setq a1 0)
         (setq a2 0))
      (t
         (setq a0 ain)
         (setq a1 a0)
         (setq a2 a1)))))
```

Figure 3.7 Multip8.mac Source File (Continued).

Figure 3.10 is a block diagram of this design approach. The MacPitts source file for this design, given in figure 3.11, defines another input port, "hin," which should be connected to the high order 8 bit partial product output of the previous stage, unless the chip is the first one in the array. In that case, "hin" is connected to ground (i.e. zero.) To further reduce area, the reset function was eliminated, because it is not in any way essential to the functioning of a multiplier used in a high throughput signal processing environment such as is envisioned for this design.

This arrangement of identical processing elements connected in a linear array to produce a pipelined result is similar in concept to the systolic array approach formulated by Kung [Ref. 18], although he was more generally concerned

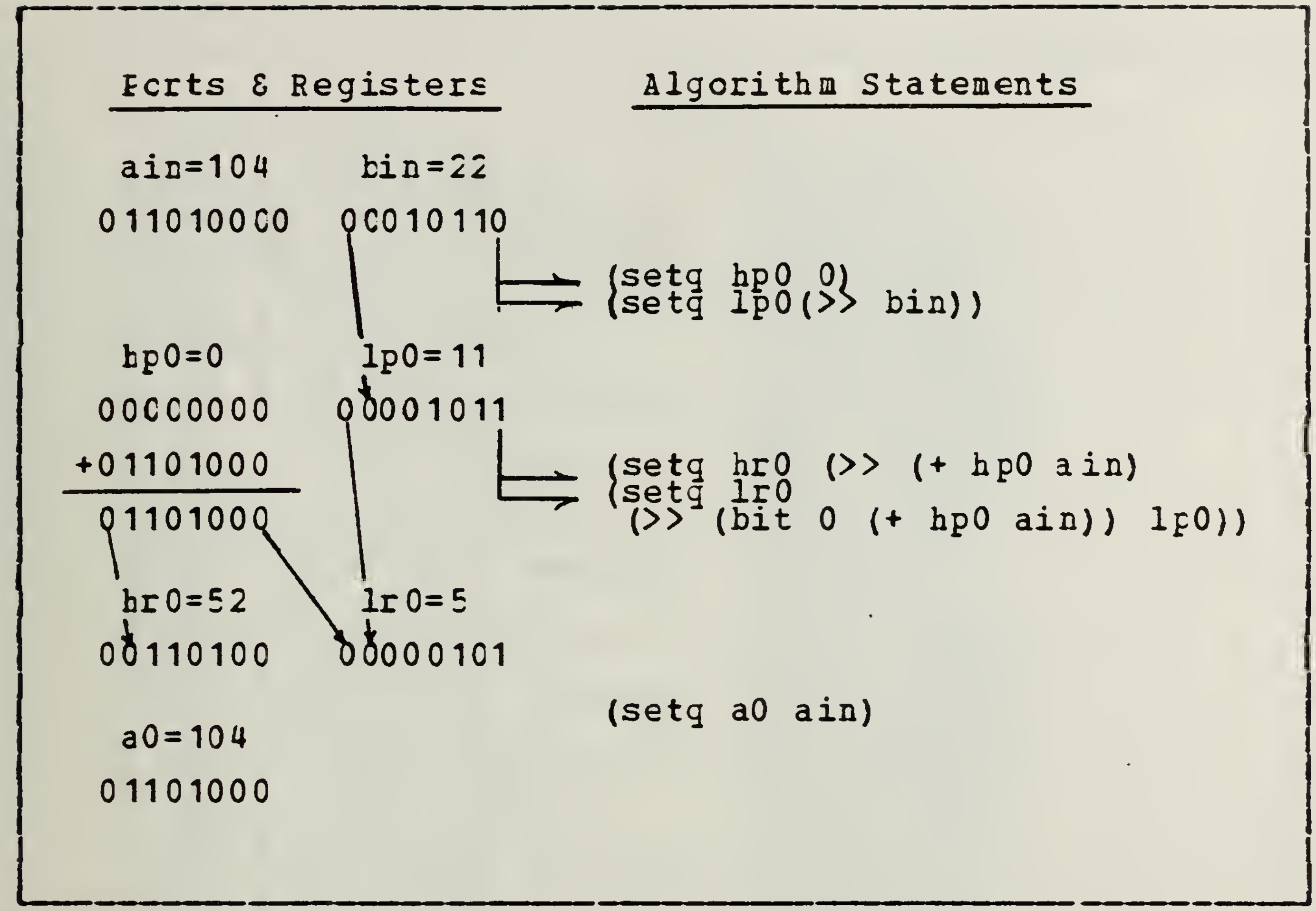


Figure 3.8 Use of Ports and Registers in multip8.mac.

with individual processing elements of greater complexity than that of multip8a cells.

The macpitts layout of multip8a has outline dimensions of 5848 x 6140 microns. The data path and control unit only occupy approximately 3000 x 2500 microns. The overall chip is large compared to its "working circuitry" because of the need to place 53 pin pads around only three sides of the perimeter. This design does not approach full utilization of the available 6890 x 6300 micron silicon area.

4. Second Partitioning: 4 Bits, 2 Stage Pipeline

It seems clear that more of the design will fit on the chip and still not exceed the maximum size for

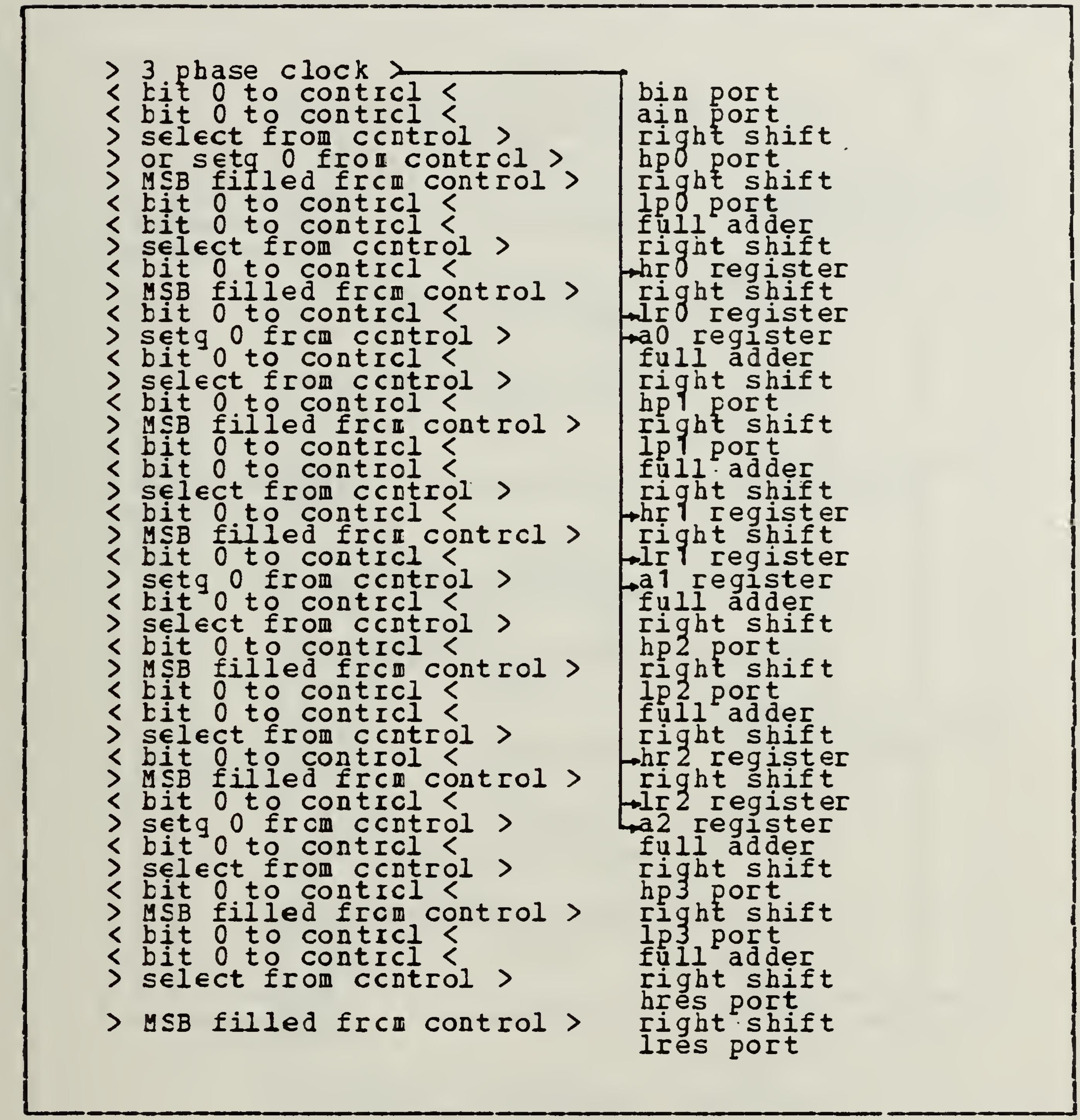


Figure 3.9 Data Path Architecture of Multip8 Chip.

fabrication. Design multip8b (source file shown in figure 3.12) tests four bits of the multiplier on one chip, therefore, only two of these chips are needed to do a complete multiplication. Essentially this is just a doubled version of multip8a. The MacPitts layout is 7130 x 6140 microns for

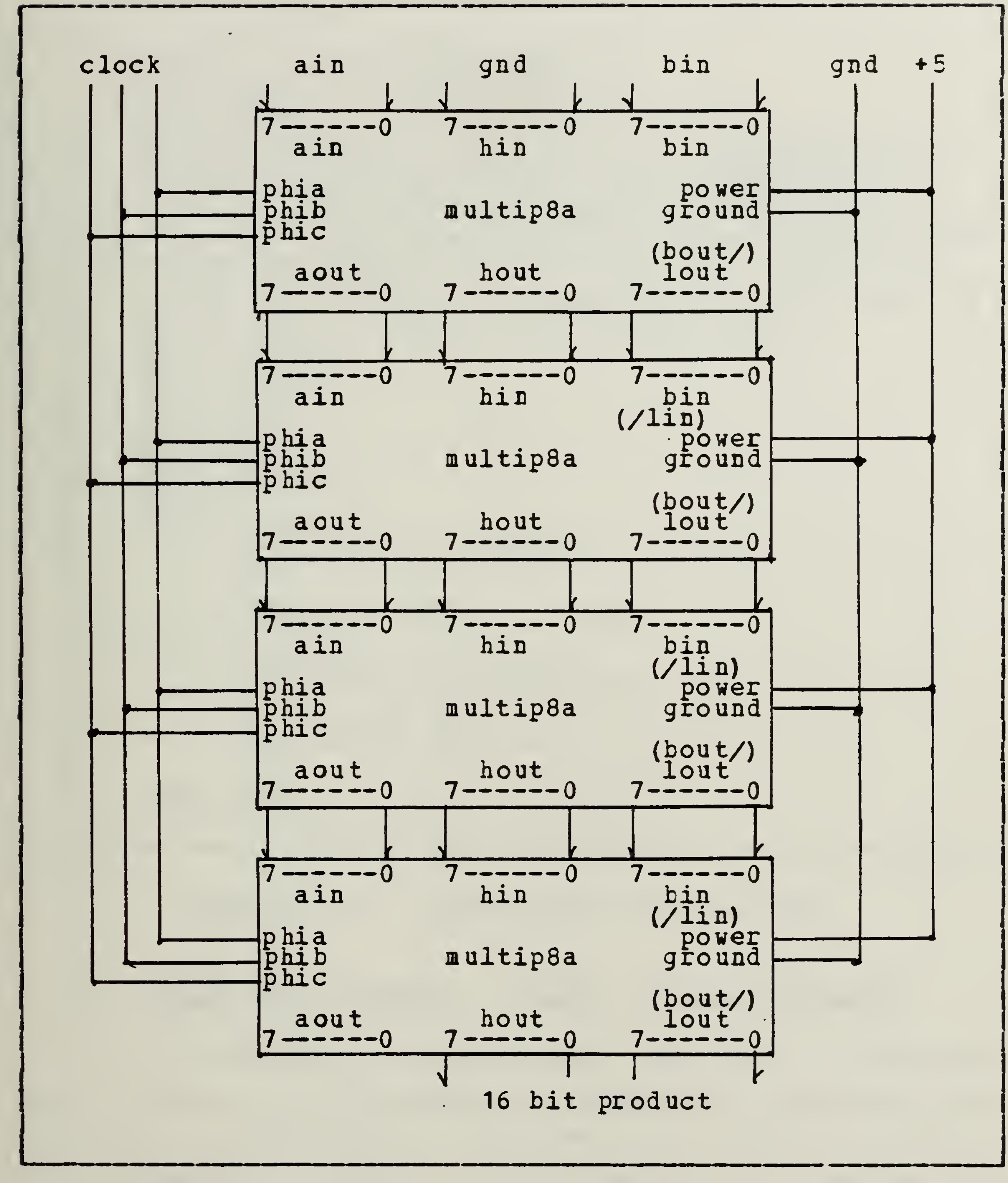


Figure 3.10 Block Diagram of First Partitioning.

the 5 micron option. This is too large to fabricate. Rerun with the 4 micron option, the multip8b chip has satisfactory dimensions: 5884 x 6024 microns.

```
; 1 stage of a 4-stage pipelined multiplier
; product is a 16 bit unsigned integer
(program multip8a 8     ; data path is 8 bits wide
  (def 1 ground)
  (def ain port input (2 3 4 5 6 7 8 9))        ;multiplicand input
  (def bin port input (10 11 12 13 14 15 16 17)) ; multiplier input
  ; this port also receives the lower 8 bits of the partial product
  (def hin port input (18 19 20 21 22 23 24 25)) ;upper 8 bits of
  ; partial product from preceding stage, zero if first stage.
  (def aout port output (26 27 28 29 30 31 32 33)) ; multiplicand output
  (def hout port output (34 35 36 37 38 39 40 41)) ; upper 8 bits of
  ; partial product output
  (def lout port output (42 43 44 45 46 47 48 49)) ; lower 8 bits of
  ; partial product output and shifted multiplier output
  (def al register)
  (def hrl register)
  (def lrl register)
  (def 50 phia)
  (def 51 phib)
  (def 52 phic)
  (def 53 power)
; end of definitions
  (always
  (cond ((bit 0 bin)
           (setq hrl (>> (+ hin ain)))
           (setq lrl (>> (bit 0 (+ hin ain)) bin)))
         (t
           (setq hrl (>> hin))
           (setq lrl (>> (bit 0 hin) bin))))
  (cond ((bit 0 lrl)
           (setq hout (>> (+ hrl ain)))
           (setq lout (>> (bit 0 (+ hrl ain)) lrl)))
         (t
           (setq hout (>> hrl))
           (setq lout (>> (bit 0 hrl) lrl))))
;
  (setq al ain)
  (setq aout al)))
```

Figure 3.11 Multip8a.mac Source File.

5. Third Partitioning: 2 Bits, 4 Stage Pipeline

By replacing every internal port with a register, and providing two additional corresponding pipeline registers for the multiplicand, the delay per pipeline stage can be reduced by a factor of approximately two because the adders drive a register directly instead of through a port and another adder. The clock rate can therefore be approximately doubled. This modification has another attractive feature in that it allows the output port to be driven

```
; 2 stages of a 4-stage pipelined multiplier
; product is a 16 bit unsigned integer
(program multip8b 8     ; data path is 8 bits wide
  (def 1 ground)
  (def ain port input (2 3 4 5 6 7 8 9))        ;multiplicand input
  (def bin port input (10 11 12 13 14 15 16 17)) ; multiplier input
  ; this port also receives the lower 8 bits of the partial product
  (def hin port input (18 19 20 21 22 23 24 25)) ;upper 8 bits of
  ; partial product from preceding stage, zero if first stage.
  (def aout port output (26 27 28 29 30 31 32 33)) ; multiplicand output
  (def hout port output (34 35 36 37 38 39 40 41)) ; upper 8 bits of
  ; partial product output
  (def lout port output (42 43 44 45 46 47 48 49)) ; lower 8 bits of
  ; partial product output and shifted multiplier output
  (def a1 register)
  (def a2 register)
  (def hr1 register)
  (def lr1 register)
  (def hp1 port internal)
  (def lp1 port internal)
  (def hr2 register)
  (def lr2 register)
  (def 50 phia)
  (def 51 phib)
  (def 52 phic)
  (def 53 power)
; end of definitions
  (always
  (cond ((bit 0 bin)
           (setq hr1 (>> (+ hin ain)))
           (setq lr1 (>> (bit 0 (+ hin ain)) bin)))
         (t
           (setq hr1 (>> hin))
           (setq lr1 (>> (bit 0 hin) bin))))
  (cond ((bit 0 lr1)
           (setq hp1 (>> (+ hr1 ain)))
           (setq lp1 (>> (bit 0 (+ hr1 ain)) lr1)))
         (t
           (setq hp1 (>> hr1))
           (setq lp1 (>> (bit 0 hr1) lr1))))
  (cond ((bit 0 lp1)
           (setq hr2 (>> (+ hp1 a1)))
           (setq lr2 (>> (bit 0 (+ hp1 a1)) lp1)))
         (t
           (setq hr2 (>> hp1))
           (setq lr2 (>> (bit 0 hp1) lp1))))
  (cond ((bit 0 lr2)
           (setq hout (>> (+ hr2 a1)))
           (setq lout (>> (bit 0 (+ hr2 a1)) lr2)))
         (t
           (setq hout (>> hr2))
           (setq lout (>> (bit 0 hr2) lr2))))
;
  (setq a1 ain)
  (setq a2 a1)
  (setq aout a2)))
```

Figure 3.12 Multip8b.mac Source File.

```
; 2 stages of a 4-stage pipelined multiplier
; product is a 16 bit unsigned integer
(program multip8c 8     ; data path is 8 bits wide
  (def 1 ground)
  (def ain port input (2 3 4 5 6 7 8 9))        ;multiplicand input
  (def bin port input (10 11 12 13 14 15 16 17)) ; multiplier input
  ; this port also receives the lower 8 bits of the partial product
  (def hin port input (18 19 20 21 22 23 24 25)) ;upper 8 bits of
  ; partial product from preceding stage, zero if first stage.
  (def aout port output (26 27 28 29 30 31 32 33)) ; multiplicand output
  (def hout port output (34 35 36 37 38 39 40 41)) ; upper 8 bits of
  ; partial product output
  (def lout port output (42 43 44 45 46 47 48 49)) ; lower 8 bits of
  ; partial product output and shifted multiplier output
  (def a1 register)
  (def a2 register)
  (def a3 register)
  (def a4 register)
  (def hr1 register)
  (def lr1 register)
  (def hr2 register)
  (def lr2 register)
  (def hr3 register)
  (def lr3 register)
  (def hr4 register)
  (def lr4 register)
  (def 50 phia)
  (def 51 phib)
  (def 52 phic)
  (def 53 power)
; end of definitions
  (always
  (cond ((bit 0 bin)
           (setq hr1 (>> (+ hin ain)))
           (setq lr1 (>> (bit 0 (+ hin ain)) bin)))
         (t
           (setq hr1 (>> hin))
           (setq lr1 (>> (bit 0 hin) bin))))
  (cond ((bit 0 lr1)
           (setq hr2 (>> (+ hr1 a1)))
           (setq lr2 (>> (bit 0 (+ hr1 a1)) lr1)))
         (t
           (setq hr2 (>> hr1))
           (setq lr2 (>> (bit 0 hr1) lr1))))
  (cond ((bit 0 lr2)
           (setq hr3 (>> (+ hr2 a2)))
           (setq lr3 (>> (bit 0 (+ hr2 a2)) lr2)))
         (t
           (setq hr3 (>> hr2))
           (setq lr3 (>> (bit 0 hr2) lr2))))
  (cond ((bit 0 lr3)
           (setq hr4 (>> (+ hr3 a3)))
           (setq lr4 (>> (bit 0 (+ hr3 a3)) lr3)))
         (t
           (setq hr4 (>> hr3))
           (setq lr4 (>> (bit 0 hr3) lr3))))
;
  (setq hout hr4)
  (setq lout lr4)
  (setq a1 ain)
  (setq a2 a1)
  (setq a3 a2)
  (setq a4 a3)
  (setq aout a4)))
```

Figure 3.13 Multip8c.mac Source File.

directly by a register rather than by an adder. Thus, the output data is valid sooner after the completion of a clock cycle than it was in the case of multip8b.

Some room to spare on the multip8b 4 micron layout leaves hope that this four stage pipeline algorithm, figure 3.13, may be feasible. In fact, the macpitts layout for multip8c measures 8218 x 6140 microns in 5 micron technology. In 4 micron technology the chip measures 6766 x 6024 microns, which consumes almost 94 per cent of the maximum allowable chip area. This is a good indication that the limit may in fact have been reached on obtaining any more elaborate design variations for the multiplier which can be fabricated by the standard MOSIS facilities.

A summary of statistics produced by macpitts for all the multiplier designs explored in this chapter is given in table I. Each line represents a different cif file, some of which may be derived from the same source file, with the only difference being the invocation options. The root of each entry in the "DESIGN" column corresponds to the name of a multiplication algorithm introduced previously in this chapter. To clarify the notation of the "DESIGN" column, note that the last digit gives the minimum feature size selected, in microns. Where no digit is explicitly stated, the minimum feature size is 5 microns.

E. DESIGN VALIDATION

1. Functional Simulation

Before proceeding with fabrication it is necessary to validate the multip8c4 design by functional simulation, design rule checking and node extraction with subsequent event simulation.

TABLE I

Statistics For MacPitts Multiplier Chip Designs

DESIGN	CHIP DIMENSIONS (Microns)	POWER DISSIPATION (Watts)	MACPITTS CPU TIME (Minutes)	NUMBER OF TRANSISTORS	NUMBER OF DATA PATH UNITS	NUMBER OF CONTROL TRACKS
multic	4320 x 2557.5	0.118015	10.6	569	16	15
multip	6320 x 2847.5	0.17212	30.4	1129	25	17
multip8	11848 x 4897.5	0.4750	54.6	3016	58	38
multip8a	5848 x 6140	0.2398	20.1	1220	16	10
multip8b5	7130 x 6140	0.34998	34.75	1999	31	19
multip8b4	5884 x 6024	0.3498	34.7	1999	31	19
multip8c5	8218 x 6140	0.407016	43.5	2497	35	21
multip8c4	6766 x 6024	0.407016	40.8	2497	35	21
multip8c4 (nocd)	6766 x 6024	0.40716	32.35	2497	35	32

Note: The (nocd) run was made with the noopt-c and noopt-d options in effect.

In any functional simulation the first issue to address is, "How exhaustive shall the simulation be?" Truly exhaustive testing of multip8c4 is a formidable task, at best. The number of different electrically possible combinations of bits for the three input ports--ain, bin and hin--is

$$(2^8)^3 = 2^{24} = 16,777,216.$$

Then, there are four internal pipeline stages. Therefore, ideally, every sequence of 4 of these 16,777,216 inputs should be tested because there should be no restrictions on the ordering of problems in the pipeline. This consideration increases the number of possible states to

$$(16,777,216)^4 = 7.92\text{x}10^{28} \text{ states.}$$

Each state transition requires five transitions of the raw clock, as will be recalled from figure 2.4. It is reasonable to assume a raw clock frequency of 10 MHz for an NMOS circuit. For the master-slave flip flops used in MacPitts this translates to a state transition rate of 2 MHz. From this assumption the time to cover all states of this circuit is calculated to be

$$7.92\text{x}10^{28} \text{ states / } 2\text{x}10^6 \text{ states/sec} = 3.96\text{x}10^{22} \text{ seconds}$$
$$3.96\text{x}10^{22} \text{ sec / } 8.64\text{x}10^4 \text{ sec/day} = 4.58\text{x}10^{17} \text{ days}$$
$$4.58\text{x}10^{17} \text{ days / } 365 \text{ days/year} = 1.26\text{x}10^{15} \text{ years}$$

Therefore testing every electrically possible state, even once, is obviously impractical.

If only each 24 bit input combination were tested once, without regard for the order in which these tests were conducted, the time required is only

$$16,777,216 \text{ / } 2\text{x}10^6 = 8.38 \text{ seconds.}$$

It should be remembered that, in its intended application, the number of expected input combinations to multip8c is considerably smaller. There are only (255x127) +1 or 32386 possible 7x8 bit multiplication problems. Each of these will have hin=0 on the first chip. The second chip will

have only one unique set of inputs passed to it by the first chip for each of these 32386 problems. Therefore, the total number of different input combinations of ain, bin and hin that will be encountered in actual operation is no greater than 2x32386 or 64772. The precise number is somewhat smaller still because some problems, such as those which have zero for the multiplier or multiplicand, will output a zero from hout in the first chip to hin of the second chip thus duplicating the first chip set of inputs for some other problem.

When using the macpitts interpreter to run a functional simulation, at least fifteen seconds must be allowed for computing the changes at each clock cycle. This fact makes testing even all expected input combinations impractical. Instead one random problem is chosen: 104 x 22 = 2288. The product 2288 is represented as hout=00001000=8, decimal and lout=11110000=240, decimal, since (256x8)+240=2288. Figures D.2 through D.6 in Appendix D show interpreter output files for each of the 8 clock cycles needed to produce the result, and a ninth clock cycle to demonstrate that the output is not subject to uncommanded changes. Between clock cycles 4 and 5 the inputs were changed to simulate two chips in cascade. The results are correct, indicating proper behavior of the specification algorithm.

A source listing for the program "values" appears in figure 3.14, together with a sample run using the problem given above. This program allows generation of the multip8c result given any combination of ain, bin and hin values entered from the terminal keyboard.

2. Design Rule Checking

The reality of the claim that MacPitts designs are "correct by construction" can be tested. The multip8c.cif

```
main()  /* interactive simultation of multip8c chip */
{
unsigned int ain, bin, hin, hout, lout, result;
unsigned int test1, test2, c;
printf ("Type ^C anytime to quit.\n\n");


/*  Loop until interupt is signaled from keyboard */
top:

/* Read input values from keyboard.  */
  printf ("Enter ain... ");
  scanf ("%d", &ain);
  printf ("Enter bin... ");
  scanf ("%d", &bin);
  printf ("Enter hin... ");
  scanf ("%d", &hin);

/* Compute the results:  first initialize output registers. */
  lout = bin;
  hout = hin;

  /* Simulate multip8c algorithm. */
  for (c=1; c<=4; c++) {
     test1 = lout & 001;
     if (test1 == 1)
       hout = hout + ain;
     lout = lout >> 1;
     test2 = hout & 001;
     if (test2 == 1)
       lout = lout + 128;
     hout = hout >> 1;
  }

  /* Put output reister values into concatenated decimal form. */
  result = 256*hout + lout;

  /* Display all values on the screen. */
  printf ("ain=%-4d bin=%-4d hin=%-4d hout=%-4d lout=%-4d result=%-5d\n\n",
                  ain,bin,hin,hout,lout,result);

  goto top;
 }


**************  SAMPLE RUN  *******************

% values
Type ^C anytime to quit.

Enter ain... 104
Enter bin... 22
Enter hin... 0
ain=104  bin=22   hin=0    hout=39   lout=1    result=9985

Enter ain... 104
Enter bin... 1
Enter hin... 39
ain=104  bin=1    hin=39   hout=8    lout=240  result=2288

Enter ain... ^C%
%
```

Figure 3.14   Values: Program to Compute Multip8c Output.

file was checked for design rule errors by running it through the Stanford "drc" program via "cll" [Ref. 1: pp. 147-151] to reformat the file. The command sequence is:

```
        % cif multip8c.cif -gnq
        % cll multip8c.co
        % drc multip8c.sco  .
```

There are two problems, however, with using drc on this design. One is that the design rules used by MacPitts are not the standard Mead Conway rules [Ref. 2: pp. 47-51], but are a combination of these and the MOSIS design rules which include burried contacts [Ref. 2: page 133]. Burried contacts are not recognized by "drc." The other problem is that the "cif" program does not correctly read .cif files which use the 200 centimicron lambda dimension--round-off error is introduced. Therefore, the design rule check can only be performed on multip8c5, not on multip8c4 which is the version to be fabricated.

The results of this drc run, thus caveated, produced 2 types of stated errors, both of which are spurious. One is a "poly to diffusion contact separation" error in the controller where macpitts abuts two contacts, one to poly and one to diffusion, but both through the same overlying metal conductor. The intent of the design rule checker, in this instance, is to forewarn of the possibility of a short circuit; a short circuit is in fact the desired result of this unorthodox structure. The other stated error is an "implant surround" error in the register clock. This structure is flagged because the burried contact to that layer was ignored by drc. Based on this non-ideal but only available check of design rules, it was concluded that the multip8c5.cif file does define processable mask layers. It is assumed that the multip8c4.cif file is also processable because it differs only in scale from multip8c5.cif, except for the pads, whose design is supposedly from a standard library supplied by MOSIS.

3. Node Extraction and Event Simulation

The node extraction program "extract" which is part of the Stanford VLSI design tools does not accurately interpret .cif files with lambda equal to 200 centimicrons. Fortunately, the "mextra" program, written at Berkeley,[2] can accommodate both 200 and 250 centimicron cif files.

To obtain an extraction and simulation of the multip8c design in 4 micron size, the corresponding cif file, multip8c4.cif was converted to the ".ca" format used by the Berkeley "caesar" layout editor. Then labels for all the pads were added to the design using caesar so that mextra would know which nodes are to be accessible for monitoring. Before exiting caesar, a new cif file, mul8c.cif, is written using the caesar command

```
: cif -p mul8c.
```

The node extraction is made by issuing the command

```
% mextra mul8c.
```

The result of the mextra run is a .sim file suitable for input to the "esim" event simulator [Ref. 1: pp. 152-155], and also a .log file (figure 3.15) in which is contained summary statistics of the extraction.

```
Window: 0 676600 0 602400
     801 depletion
    1612 enhancement
1398 nodes
```

Figure 3.15 Mextra .log File for Mul8c.cif..

---

[2] See Appendix C.

The simulation, using extracts of mul8c.cif was set up to perform the same tests used in the macpitts interpreter session of multip8c. To do this, two macro files were created. One defines the three phase clock sequence, declares which nodes to watch, and sets the values of the inputs to those which simulate the problem 104x22. The second macro file, which was designed to be read in at the midpoint of the simulation, redefines the input values to make the chip perform like the second multip8c unit in the pipeline. These files are both listed in figure 3.16.

```
% cat mul8c.macro
K phia 11011 phib 10000 phic 10001
W ain ain7 ain6 ain5 ain4 ain3 ain2 ain1 ain0
W bin bin7 bin6 bin5 bin4 bin3 bin2 bin1 bin0
W hin hin7 hin6 hin5 hin4 hin4 hin2 hin1 hin0
W hout hout7 hout6 hout5 hout4 hout3 hout2 hout1 hout0
W lout lout7 lout6 lout5 lout4 lout3 lout2 lout1 lout0
W aout aout7 aout6 aout5 aout4 aout3 aout2 aout1 aout0
W clock phia phib phic
h ain6 ain5 ain3 bin4 bin2 bin1
l ain7 ain4 ain2 ain1 ain0 bin7 bin6 bin5 bin3 bin0
l hin7 hin6 hin5 hin4 hin3 hin2 hin1 hin0

% cat mul8c.macro2
h hin5 hin2 hin1 hin0 bin0
l bin7 bin6 bin5 bin4 bin3 bin2 bin1
```

Figure 3.16 Two Macro Driver Files for Event Simulation.

The record of a simulation run using these files is contained in Appendix D. It shows the same correct results obtained with the macpitts functional interpreter. Note however that when the "I" command is given to esim, all the circuit nodes are initialized to some value over which the user has no control. Therefore, the values of the output ports are not meaningful until the fourth clock cycle, even though they are defined during initialization.

The event simulation result is encouraging evidence that macpitts can produce, in at least one instance, a mask-level description that correctly reflects a circuit design with algorithmic behavior specified by the designer. Further validation evidence was obtained by performing an extraction and event simulation on multip8c5.cif, the 5 micron version of the multiplier. This extraction could be done using the Stanford program; the result was the same as for mextra. The event simulation produced a correct result for the same exercise. It was concluded, therefore, that the design was ready for fabrication.

F. SUMMARY OF ACTIVITIES IN THE MACPITTS DESIGN CYCLE

A recommended pattern of steps to follow in the MacPitts design cycle can be summarized by presenting the sequence of UNIX commands issued by the designer for a typical case. This sequence divides into two paths after the cif file is created, depending on whether 4 micron or 5 micron minimum feature size is selected. For the 4 micron option the caesar/mextra tools must be used. For the 5 micron option it is more convenient to use extract, a program which recognizes node labels furnished by MacPitts with the cif user extension 0.

As a starting point, it is assumed that the designer already has formulated a precise idea of what behavior the chip is to exhibit, and has translated the behavioral specification into MacPitts language.

The 5 micron path, using the multip8c.mac source file as an example, is as follows:

```
% vi multip8c.mac
```

(Create the source file.)

```
% macpitts multip8c int herald
```

(Run the interpreter to debug the source file and verify the functional correctness of the specification. Save states as desired using the "p" interpreter command, renaming files from a second terminal keyboard to prevent overwriting. Quit the interpreter.)

```
% script
```

(Start a recording session for the terminal screen.)

```
% macpitts multip8c 5u herald
```

(Generate 5 micron multip8.cif and complete design statistics.)

```
% mv multip8c.cif multip8c5.cif
```

(Rename cif file to proclaim that it is a 5 micron design.)

```
% ctrl-D
```

(Stop the recording session.)

```
% print typescript
```

(Get hardcopy of compiler statistics and heralds.)

```
% cif multip8c5.cif -gnq
% cll multip8c5.co
% drc multip8c5.sco
```

(Obtain design rules check.)

```
% extract multip8c5
```

(Obtain a node extract.)

```
% vi multip8c5.sym
```

(Change spelling of VDD and ground node labels to Vdd and GND, respectively.)

```
% sim multip8c5
```

(Obtain the multip8c5.sim file.)

```
        % vi multip8c.macro1
```

(Create one or more testing sequence files for the event simulator. See the "esim" section of Appendix C for details.)

```
        % script
        % esim multip8c5.sim multip8c.macro1
```

(Perform event simulation of chip.)

```
        % ctrl-D
        % print typescript

        % vi multip8c5.cif
```

("Comment out" the user extension 0 lines at the beginning of this file by enclosing them all in one set of parentheses followed by a semicolon. See the "cifplot" section of Appendix C for details.)

```
        % stipple multip8c5.cif
```

(Obtain stipple plot on the Versatec plotter.)

The 4 micron path, using the same example, contains exactly the same steps through the interpreter run, then continues as follows:

```
        % script
        % macpitts multip8c 4u herald
```

(Generate 4 micron multip8c.cif and complete statistics.)

```
        % mv multip8c.cif multip8c4.cif
```

(Rename cif file to proclaim that it is a 4 micron design.)

```
        % ctrl-D
        % print typescript
```

```
% cif2ca multip8c4.cif
```

(Convert cif to caesar format. Benign warnings are issued when user extension 0 lines are encountered.)

```
% mv project.ca multip8c4.ca
```

(Give the top level caesar file a suitable name.)

```
% caesar multip8c4
```

(Use caesar to affix labels to each bonding pad, then output a new cif file using : cif -p cmul8c4. See the "caesar" section of Appendix C for details. Quit ceasar.)

```
% mextra cmul8c4
```

(Obtain a node extraction.)

```
% vi multip8c.macro1
```

(Testing sequence file(s) is/are identical to the 5 micron case.)

```
% script
% esim cmul8c4.sim multip8c.macro1
```

(Perform event simulation of chip.)

```
% ctrl-D
% print typescript
% stipple cmul8c4.cif
```

(Obtain stipple plot on Versatec. There is no need to worry about user extension 0 if the cif file was created by caesar.)

IV. MACPITTS PERFORMANCE

A. LAYOUT ERRORS AND INEFFICIENCIES

1. Inefficiencies

Appendix E contains photographs of an AED 767 color graphics terminal screen displaying the MacPitts chip layouts for each of the six multipliers discussed. The presentations were generated by the caesar VLSI circuit editor [Ref. 6]. Examination of these layouts, aided by the zoom-in feature of caesar, prompts several observations about MacPitts' performance.

In any VLSI circuit layout a primary goal is to cover the available silicon area as densely as possible with circuitry. A variable, but generally small amount of the silicon area within the bounding box of MacPitts layouts is covered with circuitry. This is due in part to the rigidity of the target architecture--requiring the layout of data path organelles in a strictly linear fashion. The most serious waste of space in the examples explored, however, is caused by the inability of MacPitts to install bonding pads on all four sides of the chip. The left side is never available for this purpose due to certain algorithmic simplifications made by the authors of MacPitts [Ref. 16: p. 13]. A three-sided arrangement of pads stretches the outline dimensions, particularly in designs which specify a large number of external connections. All of the partitioned multiplier algorithms presented in the previous chapter--multip8a, multip8b, and multip8c--are in this category.

One may consider the possibility of filling the large void above the useful circuitry in multip8c4, for

example, with another identical instantiation of the multip8c4 layout, minus the pads, and thereby produce a complete 8 bit multiplier on one chip. Eight pads for the hin port could then also be eliminated. The cell movement and yank/put commands of caesar would make this operation possible with a minimum of drudgery. But the interconnections between the 2 instantiations of the multip8c4 modules would still require tedious manual layout, and would be very subject to human error. Such hand crafting, minus the interconnection modifications, was, in fact, attempted. Appendix E contains a photograph displaying the results of this effort, named multip8c4d to denote "double." It clearly demonstrates that the synergistic use of MacPitts with caesar is feasible.

To pursue the manual editing approach very far would be to abandon the basic concept of silicon compilation as defined from the outset. Nevertheless, editing is required if one is to obtain efficient use of silicon resources. The appreciation of silicon compilers like MacPitts still awaits a future in which to perform such manual editing is more costly (in custom designs intended only for small volume production) than the silicon area wasted in a suboptimal layout. One can predict that that future will arrive, just as it did when the cost of memory hardware dropped thus solving an analogous problem: whether to waste memory but write clear programs, or conserve memory fully at the cost of monumental programming effort.

A lack of compactness detracts from more than economy of production, however. There are penalties in circuit operating speed as well. A closer look at the details of MacPitts layouts reveals inefficiencies which directly affect circuit performance. In general, the length of metal and polysilicon interconnections is much longer than the minimum an experienced human layout artist would be

expected to produce, even when both are limited to using right-angle (Manhattan) layout rules. For example, all of the output data bits generated at the far right side of the data path must be routed back to the left along the entire length of the data path, then up (or down), over to the right again for the entire length of the data path, and finally down (or up) again to reach the bonding pads. In the multip8c4 layout, MacPitts uses wire runs of up to 18 mm to route data bits from their sources to their bonding pads which, in some cases are less than 1 mm direct distance from the source. The problem lies in the inability of MacPitts to jump over the metal power/ground bus frame in making connections from the data path to bonding pads. This problem is illustrated in figure 4.1. The experienced user can help equalize interconnection lengths somewhat by

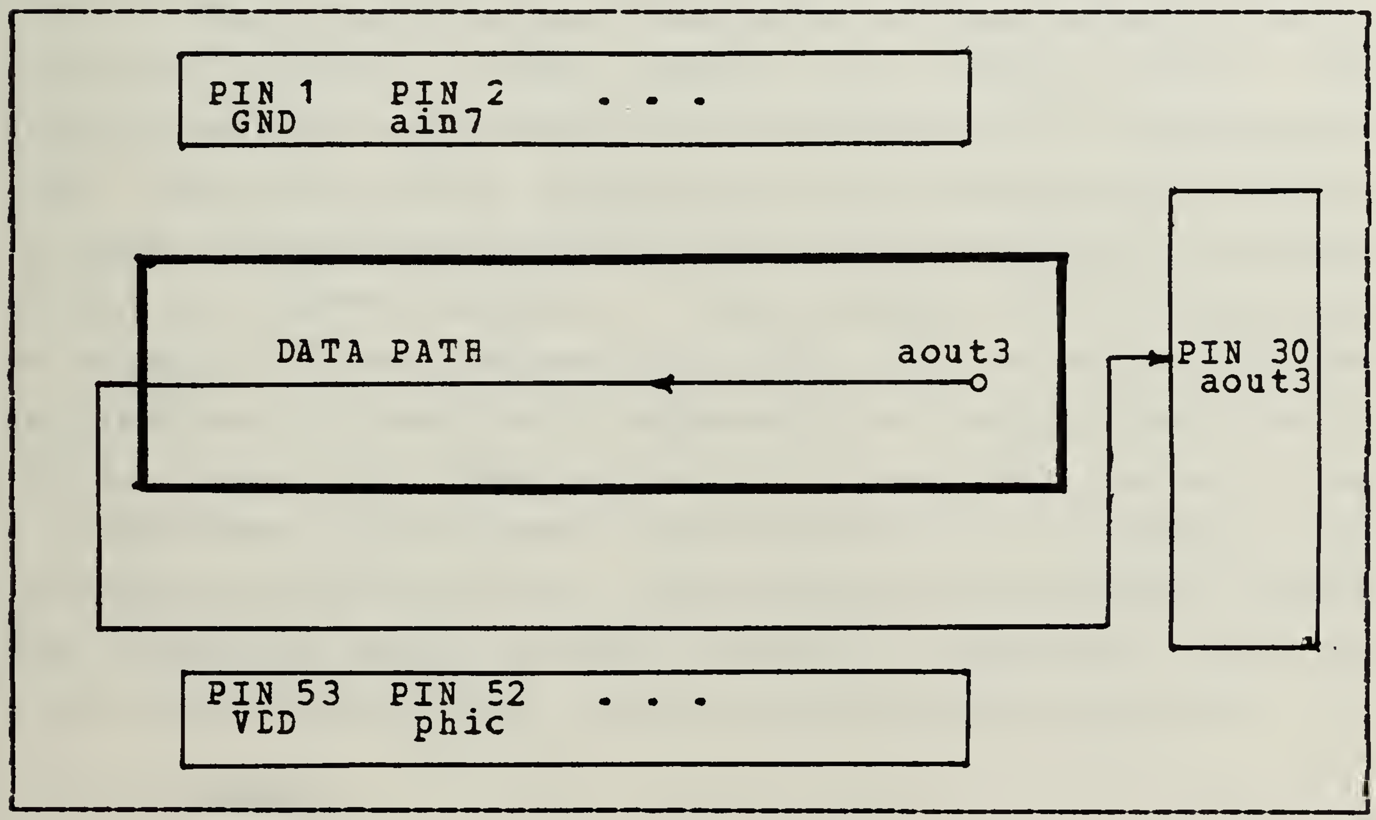


Figure 4.1 Data Path Ouput Routing.

assigning output ports only to the lowest and highest numbered pins.

MacPitts, therefore, requires that the user provide a functional specification which is enlightened by knowledge of the layout limitations if optimum performance is to be obtained. This is an area for improvement in pursuit of the silicon compiler ideal.

Another layout problem is more difficult to deal with: the excessive length of wiring between the control unit and the data path. This could be improved by centering the control unit under the data path, which would require changing the Macpitts source code in some undetermined way. As currently written, MacPitts always begins the control unit at the left margin.

There are also many instances of dead-ended wires in MacPitts layouts. These "roads to nowhere" occur when MacPitts extends runs beyond the last point of interconnection. They occur most frequently on the organelles, not all of whose capabilities may be used by the behavioral specification in a given instance. This appears to be a result of an attempt to use the same organelle for as many different applications as possible, apparently to control the size of the library. Untrimmed wires of this variety certainly add to inter-node capacitance, although not to the extent that inefficient routing does. Nevertheless, they surely reduce the operating speed of the circuit, and make operation noisier and perhaps less reliable at high frequencies.

2. Errors

In addition to the layout inefficiencies described, there is another problem with Macpitts layouts. At least one input file has been known to produce a layout containing a fatal error. Kelly [Ref. 19] attempted to use MacPitts to produce a butterfly switching element chip. His design

(called kchip2) has a much simpler data path than the multip8c pipeline multiplier, but it has a larger control unit. It also includes some finite state machine sequencer units which serve the independent processes he uses in the design. These are laid out to the right of the data path. The MacPitts designed layout of this circuit places a direct short circuit across the 3 clock bus lines. A picture of the portion of the chip where the error occurs is included in Appendix E. The problem arises because the clock bus contains "vias" where it must be extended from the data path to horizontally adjacent elements in the design. These "vias" allow the metal bus lines to cross vertical metal frame power or ground lines via a brief transition to the polysilicon layer, then back to the metal layer. MacPitts, however, apparently does not check for the presence of any intersecting vertical polysilicon runs to the control unit which may be placed at the same horizontal coordinate as the clock bus vias. None of the multip8 series of designs has any control lines entering the extreme right end of the data path. Therefore, the vias are safe, and the problem does not occur. It is interesting to note, however, that MacPitts still extends the clock bus well to the right beyond the point of last use , and includes a dead-end set of vias to jump over the data path frame, even though there is no need for that extension in the multip8 family of designs. It may be concluded from these observations that this problem is latent in all MacPitts designs, and one would do well to examine the control unit wiring in the vicinity of the clock bus at the right end of all frames. Caesar can be used, if necessary, to adjust the local wiring slightly to route the offending control line away from the clock vias.

B. ORGANELLES VS. STANDARD CELLS

This section briefly examines some comparative aspects of the Stanford standard cell approach used by Newkirk and Mathews [Ref. 20] and the organelles used in MacPitts.

Both standard cells and organelles are laid out as bit slices. It was hoped that there would be a one-to-one functional correspondence between at least some of the catalogued standard cells and the organelles which could form a basis for comparison. Unfortunately, there is very little functional correspondence, let alone structural correspondence, between the two. The standard cells contain only dynamic storage elements, and use a 2 phase clock. The MacPitts organelles use a 3 phase clock, and the only memory elements available are static master-slave flip-flop registers. The standard cells are designed for matched pitch. That is, they can be directly abutted, in many cases, to form full length words and arrays. Organelles, on the other hand, generally require some margin around them for interconnections (called "river routing") which apparently must be specifically tailored for each instantiation of the organelle.

It was hoped that at least the MacPitts adder organelle, which is simply a standard asynchronous full adder made entirely from NOR gates, could be compared with something from the standard cell library. The most similar standard cell in the catalogue is an adder/subtractor [Ref. 20: p.10], which is based on the OM2 arithmetic logic unit [Ref. 2: pp. 145-181]. This cell is much more flexible, yet also more specialized, than the MacPitts adder. It is capable of a full range of boolean operations, not just addition, as determined by the values on two 4 bit control port lines which are threaded through the cell. It also differs from the organelle in that its operation is clocked.

Although a comparison based on size hardly seems meaningful for these two dissimilar units, it is noted that the organelle measures 250 x 40 lambda units using measurements taken from actual layout plots. The standard cell adder measures 211 x 32 lambda units as specified in [Ref. 20: p. 11].

The MacPitts static register organelle has no functional parallel in the standard cell library for the reasons mentioned above. It measures 64 x 30 lambda units, excluding the clock buffer unit which contains a load enable line affecting all the bits in the same register. The standard cell dynamic shift register bit measures 88 x 24 lambda units, and contains a selector input line for each bit of the register built from these cells.

C. SOFTWARE INCOMPATIBILITIES

The authors of MacPitts have extended the CIF language to make "0" at the beginning of a line indicate that the rest of the line contains the coordinates of a node, the mask layer to which it applies, and a label name for that node. This is a useful feature with the Stanford node extraction programs which recognize this label device and use it automatically to make the node accessible to simulation programs simply by calling its name. This extension of CIF is unknown to the Berkeley VLSI tools. The latter use another CIF extension--"94"--to flag node labels.

V. CONCLUSION

A. SUMMARY

This thesis has described silicon compilers, and demonstrated how the MacPitts silicon compiler can be employed to design a digital pipelined multiplier using a partitioning concept.

Shortcomings of this silicon compiler have been found which make the results produced by it inferior in some ways to those produced by practiced designers. These shortcomings may be outweighed, for some applications, by the reduction in design time. The functional correctness of the MacPitts multiplier design has been demonstrated to the extent allowed by available simulation tools. Other MacPitts designs may contain errors which can be edited out with relative ease.

The user of MacPitts can affect the output of the compilation process in two meaningful ways. First, it may be possible to write the behavioral specification algorithm to allow partitioning of the design among more than one chip. This possibility should be explored when layout size is a problem. Second, proper assignment of pins can reduce the worst-case length of pin pad wiring.

Macpitts has been found compatible, except in a few cases, with other VLSI design tools at NPS. The caesar VLSI editor has been particularly useful, along with the cifplot stipple plotter, in gaining insight into the processes employed by MacPitts in producing a layout.

Although the final multiplier design was submitted for fabrication, unexpected delays in production schedules precluded testing the finished product as part of this research.

B. RECOMMENDATIONS

The following recommendations should be considered:

1. Test the multiplier chips, when they become available, using the event simulation macros and as many other input combinations as facilities allow. Single-cycle testing should be done before dynamic testing is undertaken using a direct memory access tester.

2. Dissect MacPitts designs with caesar, saving in separate cif files useful symbols to add to the local VLSI library. Symbols such as pad frames or entire data path units may be of interest.

3. Write new organelles for the MacPitts library. A carry-look-ahead adder would be a useful addition.

4. Enlarge the capabilities of MacPitts to produce designs in a CMOS technology. This would involve not only writing new data path organelles, but modifying the control unit architecture, as well.

5. Obtain a capability locally to handle file transfers over the ARPANET/MILNET system.

APPENDIX A

INSTALLATION OF MACPITTS ON VAX-11/780 UNDER UNIX 4.1 AND 4.2

A. INSTALLATION UNDER UNIX 4.1 OPERATING SYSTEM

MacPitts is distributed as a collection of discrete source code files written in the "C" programming language and in Franz Lisp Opus 38. Also included in this distribution are two library files containing the bonding pad layouts in CIF, and a library file containing the standard organelles. The complete list of files is given in table II These files are located in the directory /vlsi/macpit under ownership of vlsi.

All of the operations necessary to build macpitts are sequenced by the "Makefile," a feature of the UNIX operating system that directs the automatic compilation and assembly of source programs to produce large software modules.

Building an executable version of the macpitts program requires that each source file be first compiled by the "liszt" lisp compiler or the "cc" compiler, as appropriate. The pads.l file is a lisp source which is actully generated by another lisp source. The latter source, padgen.l, filters the bonding pad CIF information contained in the pinout and pads20 files, and produces pads.l, a list of bonding pad information in the standard syntax of Franz Lisp. Pads.l is then "liszt'ed" (compiled) to produce the pads.o object file. The next step of the process fast-loads all of the compiled object files, linking them together in a single lisp "environment." Finally, the default settings for all the macpitts options invoked at run time are overlayed. It is this linked lisp environment, with the

TABLE II

MacPitts Source Files

Makefile - a makefile used to build
the complete MacPitts system

l5.l - layout language used by macpitts
to generate CIF

- next 13 files are the lisp
source code for MacPitts

control.l
data-path.l - has built-in organelles
defstructs.l
extract.l
flags.l
frame.l - layout of obj file starts here
front-page.l
general.l
interpret.l - interactive interpreter
order.l
pads.l - created during "make macpitts"
prepass.l - execution starts here

padgen.l - makes pads.l from next 2 files
pinout - Stanford Cell Library pads
pad2Cb - MOSIS 2.0 micron pads

library - standard macro, function, test,
- and organelle library
organelles.l - compiled portion of organelle library

lincoln.l - the Lincoln Laboratory lisp environment
c-routines.c - interfaces to operating system

macpitts - dumped MacPitts environment

defaults set, which is finally dumped as the binary executable module: macpitts. To repeat: this entire process is performed automatically by the Makefile.

Because this dumped lisp environment embodies all the built in functions of Franz Lisp, as well as the functions of macpitts, it contains a very large number of lisp functions. To accommodate all these functions, the Franz Lisp compiler <u>must</u> be done over with new values for the parameters MAXFNS and TRENTS which set the maximum number of

functions and function table entries allowable. Also, the padgen.l file uses the "untyi" function of the Franz Lisp Opus 38 fast loader which permits insertion of a single character in the input buffer string. The "untyi" is not a part of the Franz Lisp Opus 36 source supplied with UNIX 4.1. Therefore, when Franz Lisp is remade with the new MAXFNS and TRENTS values, the "untyi" function must be added to the fast loader source code. The steps to accomplish a remake of Franz Lisp are as follows:

- In the file /usr/src/cmd/lisp/franz/sysat.c add the follwing line to the group of MK declarations:
  ```
  MK('untyi', Luntyi, lambda);
  ```
- In the file /usr/src/cmd/lisp/franz/h/lfuncs.h add the following line to the group of lispval declarations:
  ```
  lispval Luntyi();
  ```
- in the file /usr/src/cmd/lisp/franz/lam6.c append the following code segment:
  ```
   lispval
   Luntyi()
   {
     lispval port,ch;
     port = nil;
     switch(np-lbot) {
   case 2:  port = lbot[ 1].val
   case 1: ch = lbot[0].val;
   break;
   default:
   argerr('untyi');
     }
     if(TYPE(ch) != INT }
        errorh(Vermisc, "untyi: expects fixnum character",
        nil,False,),ch);
  ```

```
    }
    ungetc((int) ch->i,okport(port,okport(Vpiport->a.clb,
    stdin)));
    return(ch);
  }
```

- In the file /usr/src/cmd/lisp/franz/nfasl.c change the value of MAXFNS to 10000.

- In the file /usr/src/cmd/lisp.franz/h/structs.h change the value of TRENTS to 1024.

- Do a "make all" from the directory /usr/src/cmd/lisp.

Franz Lisp is now ready to compile MacPitts. The next step is to correct and modify the source code for Macpitts itself.

- In the file /vlsi/macpit/c-routines.c add these lines at the beginning:

```
#define VPRINT 0100
#define VPLOT 0200
#define VPRINTPLOT 0400
#define VGETSTATE (('v'<<8)|0
#define VSETSTATE (('v'<<8)|1
```

- In the same file add the following lines after line 188:

```
static int plotmd[] =  VPLOT,0,0 ;
static int prtmd[] =  VPRINT,0,0 ;
```

- In the same file change line 199 to:

```
ioctl(plotter,VSETSTATE,plotmd);
```

- In the same file change line 207 to:

```
ioctl(plotter,VSETSTATE,prtmd);
```

- In the file /vlsi/macpit/Makefile change line 5 to:

```
MacPitts = /vlsi/macpit/bin/macpitts
```

- In the same file change line 83 to:

```
(load 'interpret.l)\
```

- In the same file change line 84 to:

  ```
  (setq macpitts-directory '/vlsi/macpit)\
  ```

- In the same file change line 87 to:

  ```
  (setq option list '(opt-d opt-c stat obj cif nologo))\
  ```

- In the same file change line 94 to;

  ```
  mv macpitts $(MacPitts)
  ```

- In the file /vlsi/macpit/interpret.l change line 18 to

  ```
  (setq library (get-library ))
  ```

- In the file /vlsi/macpit/lincoln.l change line 1093 to

  ```
  (cfasl '|/vlsi/macpit/c-routines.o -lcurses -ltermcap|
  ```

After making these changes, macpitts is ready to "make." Type "make macpitts." All the files will be compiled, linked, loaded, and then dumped as a complete macpitts lisp environment. This takes about 45 minutes on a lightly loaded system. Next type "make install." This command simply moves the dumped executable module into the directory /vlsi/macpit/bin. Now type "make clean" to remove all the lisp object files that are no longer needed. The size of the macpitts executable module is 1384704 bytes. Finally, any user of macpitts should add the directory /vlsi/macpit/bin to the path list in the .login file in his home directory.

B. INSTALLATION UNDER UNIX 4.2 OPERATING SYSTEM

The macpitts generated on a UNIX 4.1 system will not run under UNIX 4.2. This is because the system calls are different. The version of Franz Lisp supplied with UNIX 4.2 is OPUS 38, which already includes the "untyi" function. Therefore it is <u>not</u> necessary to modify the sysat.c, lfuncs.h, or lam6.c files. It <u>is</u> necessary, however, to

increase the MAXFNS and TRENTS values just as in the case of a UNIX 4.1 installation. For 4.2 these parameters are found in the files /usr/src/ucb/lisp/franz/fasl.c and /usr/src/ucb/lisp/franz/h/structs.h, respectively. After making these two changes, change directories to /usr/src/ucb/lisp, enter super-user, and issue the command "lispconf." This starts up an interactive program which allows you to specify the type of machine on which Franz Lisp is being installed. The answers to the questions posed by this script will be obvious if you are using a VAX computer running UNIX 4.2. Next issue "make fast" from the same directory and the lisp system will be generated. This step takes about 2 hours on a lightly loaded machine. After this is done, issue "make install" to move the files into the standard system directories.

The 4.2 operating system also contains another bug that will prevent the macpitts interpreter from running. In the file /usr/src/usr.lib/libterm/tputs.c change OSPEED to TOSPEED everywhere it occurs. Then recompile tputs.c This is to avoid multiple definition of OSPEED in this file and in another file, /usr/src/usr.lib/libcurses/cr_tty.c.

The modifications to the MacPitts source code itself are the same as those required for a UNIX 4.1 installation, with the following exception and addition:

- In the file /vlsi/macpit/Makefile it is not necessary to change line 83. This line should remain:
  (fasl 'interpret)

- Opus 38 of Franz Lisp, unlike Opus 36, complains if parameters declared in a functional definition are not used in the definition itself. The MacPitts source code contains an instance of this malpractice. Therefore, in the file /vlsi/macpit/frame.l change line 1338 to:

(lambda (pad)

The process of "make macpitts" is done the same as for UNIX 4.1, but the results are somewhat different. Franz Lisp issues warnings during compilation whenever an expression is encountered which does not have the proper number of parameters immediately available. These warnings occur frequently when macpitts is made under UNIX 4.2. This happens because the macpitts source code is contained in many separate files, each of which may have external references that remain unresolved until the object modules are all loaded and linked together. These warnings have no effect on the quality of macpitts produced, but their delivery does consume cpu time. As a result, it takes approximately 90 minutes to "make macpitts" under UNIX 4.1. The final Macpitts executable is 1567888 bytes long in Opus 38 on 4.2. Finally, remember to add the /vlsi/macpit/bin directory to the path list in the .login file in your home directory.

APPENDIX B

INSTALLATION OF THE CAESAR VLSI EDITOR UNDER UNIX 4.1 AND 4.2

A. INSTALLATION UNDER UNIX 4.1

The caesar VLSI circuit editor is one of many programs contained in the distribution of 1983 VLSI C.A.D. tools from U. C. Berkeley. The distribution tape is loaded, in its entirety, in the directory /vlsi/berk83 under ownership of vlsi. Before installing the tools, perform the following:

1. Have the system programmers create a new user, "sleeper," with password "caesar," and home directory /vlsi/berk83/bin. Create a ".login" file in /vlsi/berk83/bin which consists of only the following two lines:

   ```
   sleeper
   logout
   ```

   This step allows the use of a graphics tablet to position the cursor in caesar, an important facility.

2. Have the system programmer create another new user, "cad" with the password close-held, and home directory /vlsi/berk83. This step resolves the many references to "~cad" which are scatered throughout the distribution tape.

3. In the file /vlsi/berk83/man/tmac.anc replace every occurrence of the string ~cad with the string /vlsi/berk83.

4. Edit the file /vlsi/berk83/lib/displays to contain only the following one line:

   ```
   /dev/tty22 /dev/tty20 std AED767
   ```

5. Edit the file /vlsi/berk83/src/caesar/config.c to replace every occurrence of the string cad with the string /vlsi/berk83.

6. in the file /vlsi/berk83/src/caesar/main.c find the single "return" statement in the procedure "OnCommand." Just before that statement, add a line containing the statement "GrFlush();".

7. In the file /vlsi/berk83/src/makewhatis.csh remove the string "man4" from line 8.

Now proceed with the installation by issuing the following commands. Allow each command to run to completion before issuing the next. Completion is indicated by the return of the system prompt, "%."

```
cd /vlsi/berk83/src/caesar
make
mv caesar /vlsi/berk83/bin/caesar
rm *.o
cd ..
src/makewhatis.csh
```

This completes the installation of caesar, mextra, cadman, and cif2ca. There are other programs on this distribution for which the foregoing procedure should have also been sufficient to achieve a satisfactory installation, but these remain untested.

Finally, any user of these tools should add the directory /vlsi/berk83/bin to the path list in the .login file of his home directory.

B. INSTALLATION UNDER THE UNIX 4.2 OPERATING SYSTEM

The Unix 4.2 operating system uses timing and interrupt calls which differ significantly from those used by Unix

4.1. Therefore, because caesar makes extensive use of these calls, the tool as installed for 4.1 will not run under 4.2. A different distribution tape has been written for the Berkeley 1983 design tools under UNIX 4.2. Installation of this distribution proceeds in the same way as the 4.1 distribution except that step 6 is unnecessary. The bug that this step corrects has already been corrected on the 4.2 distribution tape.

It is also necessary to change a line which occurs in five files in the directory /vlsi/berk83/src/caesar

```
from  #include <time.h>
to    #include <sys/time.h>
```

The five files affected are main.c, aed4.c, omega4.c, ramtek4.c and vect4.c.

Now proceed with the installation by issuing the following commands:

```
.cd /vlsi/berk83/src/caesar
make
mv caesar /vlsi/berk83/bin/caesar
rm *.c
cd ..
src/makewhatis.csh
```

Finally, add the directory /vlsi/berk83/bin to the path list in the .login file in your home directory.

APPENDIX C

MANUAL PAGES FOR BERKELEY DESIGN TOOLS

An online operator's manual exists for all of the VLSI design tools in the 1983 distribution from Berkeley. Information on the use of any of these can be made to appear on the terminal screen by issuing

```
cadman <program>
```

where <program> can be cadman, caesar, cif2ca, cifplot, esim, mextra, or any of the other programs in that distribution. Only those pages affecting tools used in this silicon compiler research are reproduced in this appendix.

Note that the cadman program is contained in the directory

```
/vlsi/berk83/bin
```

Therefore either include this directory in the search path of your ".login" file or invoke cadman by the full rooted command:

```
/vlsi/berk83/bin/cadman <program>.
```

NAME

cadman - run off section of UNIX manual

SYNOPSIS

cadman [-] [-t] [section] title ...

DESCRIPTION

Cadman is a program which prints sections of the cad manual. *Section* is an optional arabic section number, i.e. 3, which may be followed by a single letter classifier, i.e. 1m indicating a maintenance type program in section 1. It may also be ``cad'', ``new'', ``junk'', or ``public''. If a section specifier is given *cadman* looks in the that section of the cad manual for the given *titles*. If *section* is omitted, *cadman* searches all sections of the cad manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any.

If the standard output is a teletype, or if the flag - is given, then *cadman* pipes its output through *ssp*(1) to crush out useless blank lines, *ul*(1) to create proper underlines for different terminals, and through *more*(1) to stop after each page on the screen. Hit a carriage return to continue, a control-D to scroll 12 more lines when the output stops.

The -t flag causes *cadman* to arrange for the specified section to be *troff*'*ed* to the Versatec.

FILES

~cad/doc/cadman/man?/*

SEE ALSO

Programmer's manual: more(1), ul(1), ssp(1), man(1), appropos(1)

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

NAME

caesar - VLSI circuit editor

SYNOPSIS

```
caesar [ -n -g graphics_port -t tablet_port -p path -m
monitor_type -d display_type ] [ file ]
```

DESCRIPTION

Caesar is an interactive system for editing VLSI circuits at the level of mask geometries. It uses a variety of color displays with a bit pad as well as a standard text terminal. For a complete description and tutorial introduction, see the user manual "Editing VLSI Circuits with Caesar" (an on-line copy is in ~cad/doc/caesar.tblms).

Command line switches are:

-n	Execute in non-interactive mode.
-g	The next argument is the name of the port to use for communication with the graphics display. If not specified, Caesar makes an educated guess based on the terminal from which it is being run.
-t	The next argument is the name of the port to use for reading information from the graphics tablet. If not specified, Caesar makes an educated guess (usually the graphics port).
-p	The next argument is a search path to be used when opening files.
-m	The next argument is the type of color monitor being used, and is used to select the right color map for the monitor's phosphors. "std" works well for most monitors, "pale" is for monitors with especially pale blue phosphor.
-d	The next argument is the type of display controller being used. Among the display types currently understood are: AED512, UCB512 (the AED512 with special Berkeley PROMs for stippling), AED767, AED640 (an AED767 configured as 483x640 pixels), Omega440, R9400, or Vectrix.

When Caesar starts up it looks for a command file with the name ".caesar" in the home directory and processes it if it exists. Then Caesar looks for a .caesar file in the current directory and reads it as a command file if it exists. The .caesar file format is described under the long command *source*.

You generally have to log in on the color terminal under the name "sleeper" (password "caesar"). This is necessary in order for the tablet to be useable. Sleeper can be killed by typing two control-backslashes in quick succession on the color display keyboard (on the AED displays, control-backslash is gotten by typing control-shift-L.)

The four buttons on the graphics tablet puck are used in the following way:

left (white) (#2)
Move the box so that its fixed corner (normally lower-left) coincides with the crosshair position.

right (green) (#4)
Move the box's variable corner (normally upper-right) to coincide with the crosshair position. The fixed corner is not moved.

top (yellow) (#1)
Find the cell containing the crosshair whose lower-left corner is closest to the crosshair. Make that cell the current cell. If the button is depressed again without moving the crosshair, the parent of the current cell is made the current cell.

bottom (blue) (#3)
Paint the area of the box with the mask layers underneath the crosshair. If there are no mask layers visible underneath the crosshair, erase the area of the box.

SHORT COMMANDS

Short commands are invoked by typing a single letter on the keyboard. Valid commands are:

a	Yank the information underneath the box into the yank buffer. Only yank the mask layers present under the crosshair (if there are no mask layers underneath the crosshair, yank all mask layers and labels).
c	Unexpand current cell (display in bounding box form).
d	Delete paint underneath the box in the mask layers underneath the crosshair (if there are no mask layers underneath the crosshair, the delete labels and all mask layers).
e	Move the box up 1 lambda.

g	Toggle grid on/off.
l	Redisplay the information on both text and graphics screens.
q	Move the box left 1 lambda.
r	Move the box down 1 lambda.
s	Put back (stuff) all the information in the yank buffer at the current box location. Stuff only information in mask layers that are present underneath the crosshair (if there are no mask layers underneath the crosshair, stuff all mask layers plus labels).
u	Undo the last change to the layout.
w	Move the box right one lambda.
x	Unexpand all cells that intersect the box but don't contain it.
z	Zoom in so that the area underneath the box fills the screen.
C	Expand current cell so that its paint and children can be seen.
X	Expand all cells that intersect the box, recursively, until there are no unexpanded cells intersecting the box.
Z	Zoom out so that everything on current screen fills the area underneath the box.
5	Move the picture so that the fixed corner of the box is in the center of the screen.
6	Move the picture so that the variable corner of the box is in the center of the screen.
^L	Redisplay the graphics and text displays.
.	Repeat the last long command.

LONG COMMANDS

Long commands are invoked by typing a colon character (":"). The cursor will appear on the bottom line of the text terminal. A line containing a command name and parameters should be typed, terminated by return. Each line may consist of multiple commands separated by semi-colons (to use a colon

as part of a long command, precede it with a backslash). Short commands may be invoked in long command format by preceding the short command letter with a single quote. Unambiguous abbreviations for command names and parameters are accepted. The commands are:

align <scale>
Change crosshair alignment to <scale>. Crosshair position will be rounded off to nearest multiple of <scale>.

array <xsize> <ysize>
Make the current cell into an array with <xsize> instances in the x-direction and <ysize> instances in the y-direction. The spacing between elements is determined by the box x- and y-dimensions.

array <xbot> <ybot> <xtop> <ytop>
Make the current cell into an array, numbered from <xbot> to <xtop> in the x-direction and from <ybot> to <ytop> in the y-direction. The spacing between array elements is determined by the box x- and y-dimensions.

box <keyword> <amount>
Change the box by <amount> lambda units, according to <keyword>. If <keyword> is one of "left", "right", "up", or "down", the whole box is moved the indicated amount in the indicated direction. If <keyword> is one of "xbot", "ybot", "xtop", or "ytop", then one of the coordinates of the box is adjusted by the given amount. <amount> may be either positive or negative.

button <number> <x> <y>
Simulate the pressing of button <number> at the screen location given by <x> and <y> (in pixels). If <x> and <y> are omitted, the current crosshair position is used.

cif -sblpx <name> <scale>
Write out a CIF description of the layout into file <name> (use edit cell name by default; a ".cif" extension is supplied by default). <scale> indicates how many centimicrons to use per Caesar unit (200 by default). The -s switch causes no silicon (paint) to be output to the CIF file. The -b switch causes bounding boxes to be drawn for unexpanded cells. The -l causes labels to be output. The -p switch causes a CIF point to be generated for each label. The -x switch causes Caesar not to automatically expand all cells (they are expanded by default).

cload <file>

Load the colormap from <file>. The monitor type is used as default extension.

clockwise <degrees> [y]
Rotate the current cell by the largest multiple of 90 degrees less than or equal to <degrees>. <degrees> defaults to 90. If the command is followed by a "y" then the yank buffer is rotated instead of the current cell.

colormap <layers>
Print out the red, green, and blue intensities associated with <layers>.

colormap <layers> <red> <green> <blue>
Set the intensities associated with <layers> to the given values.

copycell
Make a copy of the current cell, and position it so that its lower-left corner coincides with the lower-left corner of the box.

csave <file>
Save the current colormap in <file> (the monitor type is used as default extension).

deletecell
Delete the current cell.

editcell <file>
Edit the cell hierarchy rooted at <file>. A ".ca" extension is supplied by default. If information in the current hierarchy has changed, you are given a chance to write it out.

erasepaint <layers>
For the area enclosed by the box, erase all paint in <layers>. If <layers> is omitted it defaults to "*l".

fill <direction> <layers>
<direction> is one of "left", "right", "up", or "down". The paint under one edge of the box (respectively, the right, left, bottom, or top edge) is sampled; everywhere that the edge touches paint, the paint is extended in the given direction to the opposite side of the box. <layers> selects which layers to fill; if omitted then a default of "*" is used.

flushcell
Remove the definition of the current definition from main memory and reload it from the disk version. Any

changes to the cell since it was last written are lost.

getcell <file>
This command makes an instance of the cell in <file> (a ".ca" extension is supplied by default) and positions that instance at the current box location. The box size is changed to equal the bounding box of the cell.

gridspacing
The grid is modified so that its spacings in x and y equal the dimensions of the box. The grid is set so that the box falls on grid points.

gripe
The mail program is run so that comments can be sent to the Caesar maintainer.

height <size>
The box's height is set to <size>. If <size> is preceded by a plus sign then the fixed corner is moved to set the correct height; otherwise the variable corner is moved. <size> defaults to 2.

identifycell <name>
The current cell is tagged with the instance name given by <name>. This feature is not currently supported in any useful fashion. <name> may not contain any white space.

label <name> <position>
A rectangular label is placed at the box location and tagged with <name>. <name> may not contain any white space. <position> is one of "center", "left", "right", "top", or "bottom"; it specifies where the text is to be displayed relative to the rectangle. If omitted, <position> defaults to "top".

lyra <ruleset>
The program ~cad/bin/lyra is run, and is passed via pipe all the mask features within 3L of the box. The program returns labels identifying design rule violations, and these are added to the edit cell. If <ruleset> is specified, it is passed to Lyra with the -r switch to indicate a specific ruleset. Otherwise, the current technology is used as the ruleset.

macro <character> <command>
The given long command is associated with the given character, such that whenever the character is typed as a short command then the given command is executed. This overrides any existing definition for the character. To clear a macro definition, type ":macro

<character>", and to clear all macro definitions, type ":macro"

mark <mark1> <mark2>
The box is saved in the mark given by <mark1>. <mark1> must be a lower-case letter. If <mark2> is specified, the box is changed to coincide with <mark2>.

movecell <keyword>
The current cell is moved in one of two ways, selected by <keyword>. If <keyword> is "byposition", then the cell is moved so that its lower-left corner coincides with the lower-left corner of the box. This also happens if no keyword is specified. If <keyword> is "bysize", then the cell is displaced by the size of the box (this means that what used to be at the fixed corner of the box will now be at the variable corner).

paint <layers>
The area underneath the box is painted in <layers>.

path <path>
The string given by <path> becomes the search path used during file lookups. <path> consists of directory names separated by colons or spaces. Each name should end in "/".

peek <layers>
Display all paint underneath the box belonging to <layers>, even for unexpanded cells and their descendants.

popbox <mark>
If <mark> is specified, then the box is replaced with the given mark. Otherwise the box stack is popped and the top stack element overwrites the box.

pushbox <mark>
The box is pushed onto the box stack. If <mark> is specified then it is used to overwrite the box, otherwise the box remains unchanged.

put <layers>
The yank buffer information in <layers> is copied back to the box location. If <layers> is omitted, it defaults to "*S1".

quit If any cells have changed since they were last saved on disk, the user is given a chance to write them out or abort the command. Otherwise the program returns to the shell.

reset

The graphics display is reinitialized and the colormap is reloaded.

return

The current subedit is left, and the containing edit is resumed.

savecell <name>

If <name> is specified then the current cell is given that name and written to disk under the name (a ".ca" extension is supplied by default). If <file> isn't specified then the cell is written out to the disk file from which it was read.

scroll <direction> <amount> <units>

The current view is moved in the indicated direction by the indicated amount. <direction> must be one of "left", "right", "up", or "down", <amount> is a floating-point number, and <units> is one of "screens" or "lambda". <units> defaults to "screens", and <amount> defaults to 0.5.

search <regexp>

Search labels and bounding boxes underneath the box for text matching <regexp>. See the manual entry for *ed* for a description of <regexp>. Push an entry onto the box stack for each match. Even unexpanded cells are searched.

sideways [y]

Flip the current cell sideways (i.e. about a vertical axis). If the command is followed by a "y" then the yank buffer is flipped instead of the current cell.

source <filename>

The given file is read, and each line is processed as one long command (no colons are necessary). Any line whose last character is backslash is joined to the following line.

subedit

Make the current cell the edit cell, and edit it in context.

technology <file>

Load technology information from <file>. A ".tech" extension is supplied by default.

upsidedown [y]

Flip the current cell upside down. If the command is followed by a "y" then the yank buffer is flipped

instead of the current cell.

usage <file>
Write out in <file> the names of all the files containing cell definitions used anywhere in the design hierarchy.

view <mark>
If <mark> is specified, set view to it, otherwise, change the view to encompass the entire edit cell.

visiblelayers <layers>
Set the visible layers to include just <layers>. Preface <layers> with a plus or minus sign to add to or remove from the currently visible ones.

width <size>
Set the box width to <size> (default is 2). Move variable corner unless width is preceded by "+", else move fixed corner.

writeall
Run through interactive script to write out all cells that have been modified.

yank <layers>
Save in the yank buffer all information underneath the box in <layers>. <layers> defaults to "*1".

ycell <name>
If <name> is specified, do the equivalent of ":getcell <name>". Then expand current cell, yank it, delete the cell, and put back everything that was yanked. This flattens the hierarchy by one level.

ysave <name>
Save the yank buffer contents in a cell named <name>. A ".ca" extension is provided by default.

LAYERS

nMOS mask layers are:

p or r
Polysilicon (red) layer.

d or g
Diffusion (green) layer.

m Metal (blue) layer.

i or y

Implant (yellow) layer.

b Buried contact (brown) layer.

c Contact cut layer.

o Overglass hole (gray) layer.

e Error layer: used by design rule checkers and other programs.

CMOS P-well mask layers are (using technology cmos-pw):

p or r
Polysilicon (red) layer.

d or g
Diffusion (green) layer.

m Metal (blue) layer.

c Contact cut layer.

P or y
P+ implant (pale yellow) layer.

w P-well (brown stipple) layer.

o Overglass hole (gray) layer.

e Error layer: used by design rule checkers and other programs.

Predefined system layers are:

* All mask layers.

l Label layer.

S Subcell layer.

C Cursor layer.

G Grid layer.

B Background layer.

SYSTEM MARKS

C The bounding box of the current cell.

E The bounding box of the edit cell.

P	The previous view.
R	The bounding box of the root cell.
V	The current view.

FILES

~cad/new/caesar, ~cad/doc/caesar.tblms

SEE ALSO

cif2ca(1)

NAME

cif2ca - convert CIF files to CAESAR files

SYNOPSIS

```
cif2ca [ -l lambda ] [ -t tech ] [ -o offset ] ciffile
```

DESCRIPTION

cif2ca accepts as input a CIF file and produces a CAESAR file for each defined symbol. Specifying the -l *lambda* option scales the output to *lambda* centi-microns per lambda. The default scale is 200 centi-microns per lambda. The -t *tech* option causes layers from the specified technology to be acceptable. The default technology is nmos. For a list of acceptable technologies, see *caesar* (1). The -o *offset* option causes all CIF numbers to be incremented by *offset*. This is useful when the CIF numbers are used for Caesar file names, and when several CIF files with overlapping numbers are to be joined together in Caesar.

Each symbol defined in the CIF file creates a CAESAR file. By default, the files are named ``symbol*m*.ca'', where *m* is the CIF symbol number (as modified by the -o *offset*). Symbols can also be named with a user-extension ``9'' command, giving a name to the symbol definition which encloses it. CIF commands which appear outside of symbol definitions are gathered into a symbol called, by default, ``project'', and are output to the CAESAR file ``project.ca''.

SEE ALSO

caesar(1)

DIAGNOSTICS

Diagnostics from *cif2ca* are supposed to be self-explanatory. Each diagnostic gives the line number from the input file, an error class (informational, warning, fatal, or panic), the error message, and the action taken by *cif2ca*, usually to ignore the CIF command. Informational messages usually refer to limitations of *cif2ca*. Warning messages usually refer to inconsistencies in the CIF file, these will typically result in CAESAR files which do not accurately reflect the input CIF file. Fatal messages refer to fatal inconsistencies or errors in the CIF file. A fatal error terminates *cif2ca* processing. Panic messages refer to internal problems with *cif2ca*. If any diagnostics are produced, a summary of the diagnostics is produced.

BUGS

``Delete Definitions'' commands are not implemented. *cif2ca* also has certain restrictions due to restrictions of CAESAR: *e.g.* non-manhattan objects are not allowed.

Library cells are not automagically included.

Some care should be taken in naming symbols, since symbol names are used for CAESAR file names. Names which are not unique in the first 14 characters will attempt to create the same CAESAR file, and only the last one wins. Similarly, one should avoid trying to have two project.ca files in the same directory.

NAME
cifplot - CIF interpreter and plotter

SYNOPSIS
cifplot [options] file1.cif [file2.cif ...]

DESCRIPTION

Cifplot takes a description in Cal-Tech Intermediate Form (CIF) and produces a plot. CIF is a low-level graphics language suitable for describing integrated circuit layouts. Although CIF can be used for other graphics applications, for ease of discussion it will be assumed that CIF is used to describe integrated circuit designs. Cifplot interprets any legal CIF 2.0 description including symbol renaming and Delete Definition commands. In addition, a number of local extensions have been added to CIF, including text on plots and include files. These are discussed later. Care has been taken to avoid any arbitrary restrictions on the CIF programs that can be plotted.

To get a plot call cifplot with the name of the CIF file to be plotted. If the CIF description is divided among several files call cifplot with the names of all files to be used. Cifplot reads the CIF description from the files in the order that they appear on the command line. Therefore the CIF End command should be only in the last file since cifplot ignores everything after the End command. After reading the CIF description but before plotting, cifplot will print a estimate of the size of the plot and then ask if it should continue to produce a plot. Type y to proceed and n to abort. A typical run might look as follows:

```
% cifplot lib.cif sorter.cif
Window -5700 174000 -76500 168900
Scale: 1 micron is 0.004075 inches
The plot will be 0.610833 feet
Do you want a plot?  y
```

After typing y cifplot will produce a plot on the Benson-Varian (11 inch Versatec) plotter.

Cifplot recognizes several command line options. These can be used to change the size and scale of the plot, change default plot options, and to select the output device. Several options may be selected. A dash(-) must precede each option specifier. The following is a list of options that may be included on the command line:

-w xmin xmax ymin ymax
(window) The -w options specifies the window; by default the window is set to be large enough to contain the entire plot. The windowing commands lets you plot

just a small section of your chip, enabling you to see it in better detail. *Xmin*, *xmax*, *ymin*, and *ymax* should be specified in CIF coordinates.

-s *float*
(scale) The -s option sets the scale of the plot. By default the scale is set so that the window will fill the whole page. *Float* is a floating point number specifying the number of inches which represents 1 micron. A recommended size is 0.02.

-l *layer list*
(layer) Normally all layers are plotted. The -l option specifies which layers NOT to plot. The *layer list* consists of the layer names separated by commas, no spaces. There are some reserved names: allText, bbox, outline, text, pointName, and symbolName. Including the layer name allText in the list suppresses the plotting of text; bbox suppresses the bounding box around symbols. outline suppresses the thin outline that borders each layer. The keywords text, pointName, and symbolName suppress the plotting of certain text created by local extension commands. text eliminates text created by user extension 2. pointName eliminates text created by user extension 94. symbolName eliminates text created by user extension 9. allText, pointName, and symbolName may be abbreviated by at, pn, and sn repectively.

-c *n*
(copies) Makes *n* copies of the plot. Works only for the Varian and Versatec. Default is 1 copy.

-d *n*
(depth) This option lets you limit the amount of detail plotted in a hierarchically designed chip. It will only instanciate the plot down *n* levels of calls. Sometimes too much detail can hide important features in a circuit.

-g *n*
(grid) Draw a grid over the plot with spacing every *n* CIF units.

-h (half) Plot at half normal resolution. (*Not yet implemented*.)

-e (extensions) Accept only standard CIF. User extensions produce warnings.

-I (non-Interactive) Do not ask for confirmation. Always plot.

-L	(List) Produce a listing of the CIF file on standard output as it is parsed. Not recommended unless debugging hand-coded CIF since CIF code can be rather long.
-a *n*	(approximate) Approximate a roundflash with an *n*-sided polygon. By default *n* equals 8. (I.e. roundflashes are approximated by octagons.) If *n* equals 0 then output circles for roundflashes. (It is best not to use full circles since they significantly slow down plotting.) (*Full circles not yet implemented*.)
-b "*text*"	(banner) Print the text at the top of the plot.
-C	(Comments) Treat comments as though they were spaces. Sometimes CIF files created at other universities will have several errors due to syntactically incorrect comments. (I.e. the comments may appear in the middle of a CIF command or the comment does not end with a semicolon.) Of course, CIF files should not have any errors and these comment related errors must be fixed before transmitting the file for fabrication. But many times fixing these errors seems to be more trouble than it is worth, especially if you just want to get a plot. This option is useful in getting rid of many of these comment related syntax errors.
-r	(rotate) Rotate the plot 90 degrees.
-V	(Varian) Send output to the varian. (This is the default option.)
-W	(Wide) Send output directly to the versatec. (Not available at NPS.)
-S	(Spool) Store the output in a temporary file then dump the output quickly onto the Versatec. Makes nice crisp plots; also takes up a lot of disk space.
-T	(Terminal) Send output to the terminal. (Not yet fully implemented.)
-Gh	
-Ga	(Graphics terminal) Send output to terminal using it's graphics capablities. -Gh indicates that the terminal is an HP2648. -Ga indicates that the terminal is an AED 512.
-X *basename*	(eXtractor) From the CIF file create a circuit

description suitable for switch level simulation. It creates two files: *basename*.sim which contains the circuit description, and *basename*.node which contains the node numbers and their location used in the circuit description.

When this option is invoked no plot is made. Therefore it is advisable not to use any of the other options that deal only with plotting. However, the *window*, *layer*, and *approximate* options are still appropriate. To get a plot of the circuit with the node numbers call *cifplot* again, without the -X option, and include *basename*.nodes in the list of CIF files to be plotted. (This file must appear in the list of files before the file with the CIF End command.)

-c *n*

(copies) The -c specifies the number of copies of the plot you would like. This allows you to get many copies of a plot with no extra computation.

-P *pattern file*

(Pattern) The -P option lets you specify your own layers and stipple patterns. *Pattern file* may contain an arbitrary number of layer descriptors. A layer descriptor is the layer name in double quotes, followed by 8 integers. Each integer specifies 32 bits where ones are black and zeroes are white. Thus the 8 integers specify a 32 by 8 bit stipple pattern. The integers may be in decimal, octal, or hex. Hex numbers start with '0x'; octal numbers start with '0'. The CIF syntax requires that layer names be made up of only uppercase letters and digits, and not longer than four characters. The following is example of a layer description for poly-silicon:

```
"NP" 0x08080808  0x04040404  0x02020202  0x01010101
     0x80808080  0x40404040  0x20202020  0x10101010
```

-F *font file*

(Font) The -F option indicates which font you want for your text. The file must be in the directory '/usr/lib/vfont'. The default font is Roman 6 point. Obviously, this option is only useful if you have text on your plot.

-O *filename*

(Output) After parsing the CIF files, store an equivalent but easy to parse CIF description in the specified file. This option removes the include and array commands (see next section) and replaces them with equivalent standard CIF statements. The resulting

file is suitable for transmission to other facilities for fabrication.

In the definition of CIF provisions were made for local extensions. All extension commands begin with a number. Part of the purpose of these extensions is to test what features would be suitable to include as part of the standard language. But it is important to realize that these extensions are not standard CIF and that many programs interpreting CIF do not recognize them. If you use these extensions it is advisable to create another CIF file using the -O options described above before submitting your circuit for fabrication. The following is a list of extensions recognized by *cifplot*.

0I *filename*;
(Include) Read from the specified file as though it appeared in place of this command. Include files can be nested up to 6 deep.

0A *s* *m* *n* *dx* *dy* ;
(Array) Repeat symbol *s* *m* times with *dx* spacing in the x-direction and *n* times with *dy* spacing in the y-direction. *s*, *m*, and *n* are unsigned integers. *dx* and *dy* are signed integers in CIF units.

1 *message*;
(Print) Print out the message on standard output when it is read.

2 "*text*" *transform* ;

2C "*text*" *transform* ;
(Text on Plot) *Text* is placed on the plot at the position specified by the transformation. The allowed transformations are the same as the those allowed for the Call command. The transformation affects only the point at which the beginning of the text is to appear. The text is always plotted horizontally, thus the mirror and rotate transformations are not really of much use. Normally text is placed above and to the right of the reference point. The 2C command centers the text about the reference point.

9 *name*;
(Name symbol) *name* is associated with the current symbol.

94 *name* *x* *y*;

94 *name* *x* *y* *layer*;
(Name point) *name* is associated with the point (*x*, *y*).

Any mask geometry crossing this point is also associated with *name*. If *layer* is present then just geometry crossing the point on that layer is associated with *name*. For plotting this command is similar to text on plot. When doing circuit extraction this command is used to give an explicit name to a node. *Name* must not have any spaces in it, and it should not be a number.

USE WITH MACPITTS CIF

The lines starting with user extension 0, which MacPitts places at the beginning of every CIF file, must either be removed or "commented out" by enclosing them in an all-encompassing set of parentheses, thus: "(....);".

MacPitts CIF files are usually very long. It has been found most convenient to run MacPitts cifplots in the background with the non-Interactive mode selected. A convenient way to do this is by using the "stipple" command:

```
     stipple filel.cif
```

FILES

~cad/.cadrc
~/.cadrc
~cad/bin/vdump (only in 4.1 BSD UNIX)
~cad/bin/stipple
/usr/lib/vfont/R.6
/usr/tmp/#cif*

ALSO SEE

mcp(cad1), vdump(cad1), cadrc(cad5)
A Guide to LSI Implementation by Hon and Sequin, Second Edition (Xerox PARC, 1980) for a description of CIF.

BUGS

The -r is somewhat kludgy and does not work well with the other options. Space before semi-colons in local extensions can cause syntax errors.

The -O option produces simple cif with no scale factors in the DS commands. Because of this you must supply a scale factor to some programs, such as the -l option to *cif2ca*.

NAME

esim - event driven switch level simulator

SYNOPSIS

```
esim [file1 [file2 ...]]
```

DESCRIPTION

Esim is an event-driven switch level simulator for NMOS transistor circuits. *Esim* accepts commands from the user, executing each command before reading the next. Commands come in two flavors: those which manipulate the electrical network, and those to direct the simulation. Commands have the following simple syntax:

```
c arg1 arg2 ... argn <newline>
```

where 'c' is a single letter specifying the command to be performed and the *argi* are arguments to that command. The arguments are separated by spaces (or tabs) and the command is terminated by a <newline>.

To run *esim* type

```
esim file1 file2 ...
```

Esim will read and execute commands, first from *file1*, then *file2*, etc. If one of the file names is preceded by a '-', then that file becomes the new output file (the default output is stdout). For example,

```
esim f.sim -f.out g.sim
```

This would cause *esim* to read commands from *f.sim*, sending output to the default output. When *f.sim* was exhausted, *f.out* would become the new output file, and the commands in *g.sim* executed.

After all the files have been processed, and if the "q" command has not terminated the simulation run, *esim* will accept further commands from the user, prompting for each one like so:

```
sim>
```

The user can type individual commands or direct *esim* to another file using the "@" command:

```
sim> @  patchfile.sim
```

This command would cause *esim* to read commands from "patchfile.sim", returning to interactive input when the file was exhausted.

It is common to have an initial network file prepared by a node extractor with perhaps a patch file or two prepared by hand. After reading these files into the simulator, the user would then interactively direct *esim*. This could be accomplished as follows:

```
esim file.sim patch.1 patch.2
```

After reading the files, *esim* would prompt for the first command. Or we could have typed:

```
% esim file.sim
```

```
sim> @  patch.1
sim> @  patch.2
```

Network Manipulation Commands

The electrical network to be simulated is made up of enhancement and depletion mode transistors interconnected by nodes. Components can be added to the network with the following commands:

e gate source drain
e gate source drain length width key xpos ypos area

Adds enhancement mode transistor to network with the specified gate, source, and drain nodes. The longer form includes size and location information as provided by the node extractor -- when making patches the short form is usually used.

d gate source drain
d gate source drain length width key xpos ypos area

Like "e" except for depletion mode devices.

C node1 node2 cap

Increase the capictance between *node1* and *node2* by *cap*. *Esim* ignores this unless either *node1* or *node2* is GND.

= node name1 name2 name3

Allows the user to specify synonyms for a given node. Used by the node extractor to relate user-provided node names to the node's internal name (usually just a number).

| comment...

Lines beginning with vertical bar are treated as comments and ignored -- useful for deleting pieces of network in node extractor output files.

i node

Input record -- output by node extractor and not used by *esim*.

Currently, there is no way to remove components from the network once they have been added. You must go back the input files and modify them (using the comment character) to exclude those components you wished removed. "N" records need not be included for new nodes the user wishes to patch into the network.

Simulator Commands

The user can specify which nodes are to have there values displayed after each simulation step:

w node1 -node2 node3 ...

Watch node1 and node3, stop watching node2. At the end of a simulation step, each watched node will displayed like so:

```
node1=0 node3=X ...
```

To remove a node from the watched list, preface

its name with a '-' in a "w" command.

W label node1 node2 ... noden

Watch bit vector. The values of nodes node1, ..., noden will displayed as a bit vector:

```
label=010100   20
```

where the first 0 is the value of node1, the first 1 the value of node2, etc. The number displayed to right is the value of the bit vector interpreted as a binary number; this is omitted if the vector contains an X value. There is no way to unwatch a bit vector.

Before each simulation step the user can force nodes to be either high (1) or low (0) inputs (an input's value cannot be changed by the simulator!):

h node1 node2 ..

Force each node on the argument list to be a high input. overrides previous input commands if necessary.

l node1 node2 ...

Like "h" except forces nodes to be a low input.

x node1 node2 ...

Removes nodes from whatever input list they happen to be on. The next simulation step will determine their correct value in the circuit. This is the default state of most nodes. Note that this does not force nodes to have an "X" value -- it simply removes them from the input lists.

The current value of a node can be determined in several ways:

v

View. prints the values of all watched nodes and nodes on the high and low input lists.

? node1 node2 ...

Prints a synopsis of the named nodes including their current values and the state of all transistors that affect the value of these nodes. This is the most common way of wondering through the network in search of what went wrong...

! node1 node2 ...

For each node in the argument list, prints a list of transistors controlled by that node.

"?" and "!" allow the user to go both backwards and forwards through the network in search of that piece causing all the problems.

The simulator is invoked with the following commands:

s

Simulation step. Propogates new values for the inputs through the network, returns when the network has settled. If things don't settle, command will never terminate -- try the "w" and "D" commands to narrow down the problem.

c

Cycle once through the clock, as define by the K command.

I

Initialize. Circuits with state are often hard to initialize because the initial value of each node is X. To cure this problem, the I command finds each node whose value is charged-X and changes it to charged-0, then runs a simulation step. If one iterates the I command a couple times, this often leads to a stable initialized condition (indicated when an I command takes 0 events, i.e., the circuit is stable).

Try it -- if circuit does not become stable in 3 or 4 tries, this command is probably of no use.

Miscellaneous Commands

D

toggle debug switch. useful for debugging simulator and/or circuit. If debug switch is on, then during simulation step each time a watched node is encountered in some event, that fact is indicated to the user along with some event info. If a node keeps appearing in this prinout, chances are that its value is oscillating. Vice versa, if your circuit never settles (ie., it oscillates) , you can use the "D" and "w" commands to find the node(s) that are causing the problem.

> filename

write current state of each node into specified file. useful for make a break point in your simulation run. Only stores values so isn't really useful to "dump" a run for later use -- see "<" command.

< filename

read from specified file, reinitializing the value of each node as directed. Note that network must already exist and be identical to the network used to create the dump file with the ">" command. These state saving commands are really provided so that complicated initializing sequences need only be simulated once.

L

invokes network processor that finds all subnets corresponding to simple logic gates and converts them into form that allows faster simulation. Often it does the right thing, leading to a 25% to 50% reduction is the time for a single step. [We know of one case where the transformation was not transparent, so caveat simulee...]

X ...
: call extension command -- provides for user extensions to simulator.

q
: exit to system.

Local Extensions

V node vector
: Define a vector of inputs for the node. The first element is initially set as the input for node. Set the next element of the vector as the input after a cycle.

R n
: Run the simulator through n cycles. If n is not present make the run as long as the longest vector. All watch nodes are reported back as vectors.

N
: Clear all previously defined input vectors.

K node1 vector1 node2 vector2 ... nodeN vectorN
: Define the clock. Each cycle, nodes 1 through N must run through their respective vectors.

SEE ALSO

mextra(CAD1)

NAME

mextra - Manhattan Circuit Extractor

SYNOPSIS

mextra [-gho] [-u _scale_] basename

DESCRIPTION

Mextra reads an intergrated circuit layout description in Caltech Intermediate Form (CIF) and creates a circuit description. From this circuit description various electical checks can be done on your circuit. The circuit description is directly compatible with _esim_, _moserc_, and _powest_.

Names

Mextra uses the CIF label construct to implement node names and attributes. The form of the CIF label command is as follows:

```
     94  name  x  y [layer];
```

This command attaches the label to the mask geometry on the specified layer crossing the point (x, y). If no layer is present then any geometry crossing the point is given the label. _Mextra_ does not recognize the CIF user extension "0" which is used by MIT and Lincoln Labs programs (eg. macpitts) to indicate node labels.

Mextra interprets these labels as node names. These names are used to describe the extracted circuit. When no name is given to a node, a number is assigned to the node. A label may contain any ASCII character except space, tab, newline, double quote, comma, semi-colon, and parenthesis. To avoid conflict with extractor generated names, names should not be numbers or end in '#_n_' where _n_ is a number.

A problem arises when two nodes are given the same name although they are not connected electrically. Sometimes we want these nodes to have the same names, other times we don't. This frequently happens when a name is specified in a cell which is repeated many times. For instance, if we define a shift register cell with the input marked 'SR.in' then when we create an 8 bit shift register we could have 8 nodes names 'SR.in'. If this happens it would appear as though all 8 of the shift register cells were shorted together. To resolve this the extractor recognizes three different types of names: _local_, _global_, and _unspecified_. Any time a local name appears on more than one node it is appended with a unique suffix of the form '#_n_' where _n_ is a number. The numbers are assigned in scanline order and starting at 0. In the shift register example, the names would be 'SR.in#0' through 'SR.in#7'. Global names do not have suffixes appended to them. Thus unconnected nodes with

global names will appear connected after extraction. (The -g causes the extractor to append unique suffixes to unconnected nodes with the same global name.) Names are made local by ending them with a sharp sign, '#'. Names are global if they end with an exclamation mark, '!'. These terminating characters are not considered part of the name, however. Names which do not end with these characters are considered unspecified. Unspecified names are treated similar to locals. Multiple occurrences are appended with unique suffixes. By convention, unspecified names signify the designer's intention that this name is a local name, but is connected to only one node. It is illegal to have a name that is declared two different types. The extractor will complain if this is so and make the name local.

Optionally *mextra* will expand local and unspecified node names with the path name of the symbol instances through which they were called. By using the -h option *mextra* will produce node names of the form:
    /*call1*/*call2*/.../*callN*/*node-name*
where *callN* is the name of the symbol instance which contains the label *node-name*, *call*N-1 is the name of the instance which contains *callN*, and so on. Named symbol instances take the following form in CIF:
    91 *name*; C *number* [*a* *b*];
Unnamed CIF calls are assigned names of the form '#n', where n is a number.

It makes no difference to the extractor if the same name is attached to the same node several times. However, if more than one name is given to a node then the extractor must choose which name it will use. Whenever two names are given to the same node the extractor will assign the name with the highest type priority, global being the highest, unspecified next, local lowest. If the names are the same type then the extractor takes the one with the fewest slashes('/'); if the number of slashes is equal, the shortest name is taken. This causes the name highest up in the symbol hierarchy to be taken when hierarchical names are expanded. At the end of the log file the extractor lists nodes with more than one name attached. These lines start with an equal sign and are readable by *esim* so that it will understand these aliases.

Attributes

In addtion to naming nodes *mextra* allows you to attach attributes to nodes. There are two types of attributes, *node attributes*, and *transistor attributes*. A node attribute is attached to a node using the CIF 94 construct, just the same way as a node name. The node attribute must end in an at-sign, '@'. More than one attribute may be attached to a node. *Mextra* does not interpret these attributes other

than to eliminate duplicates. For each attribute attached to a node there appears a line in the .sim file in the following form:

A *node* *attribute*

Node is the node name, and *attribute* is the attribute attached to that node with the at-sign removed.

Transistor attributes can be attached to the gate, source, or drain of a transistor. Transistor attributes must end in a dollar sign, '$'. To attach an attribute to a transistor gate the label must be placed inside the transistor gate region. To attach an attribute to a source or drain of a transistor the label must be placed on the source or drain edge of a transistor. Transistor attributes are recorded in the transistor record in the .sim file. A transistor description has the following form:

type *gate* *source* *drain* *l* *w* *x* *y* g=*attributes* s=*attributes* d=*attributes*

Attributes is a comma-separated list of attributes. If no attribute is present for the gate, source, or drain the g=, s=, or d= fields may be omitted.

Capacitance

The .sim file also has information about capacitance in the circuit. The lines containing capacitance information are of the form:

C *node1* *node2* *cap-value*

cap-value is the capacitance betweens the nodes in femto-farads. Capacitance values below a certain threshold are not reported. The default threshold is 50 femto-farads.

The extractor reports capacitance from two sources - capacitance between node and substrate, and capacitance caused by poly overlapping diffusion but not forming a transistor. Transistor capacitances are not included since most of the tools that work on the .sim file calculate the transistor capacitance from the width and length information.

The capacitance for each layer is calculated separately. The reported node capacitance is the total of the layer capacitances of the node. The layer capacitance is calculated by taking the area of a node on that layer and multiplying it by a constant. This is added to the product of the perimeter and a constant. The default constants are given below. Area constants are in femto-farads per square micron. Perimeter constants are femto-farads per micron.

layer	area	perimeter
metal	0.03	0.0
poly	0.05	0.0

diff	0.1	0.1
poly/diff	0.4	0.0

Poly/diffusion capacitance is calculated similar to layer capacitance. The area is multiplied by constant and this is added to the perimeter multiplied by a constant. Poly/diffusion capacitance is not threshold, however.

The -o option supresses the calculation of capacitance, and instead, gives for each node in the circuit the area and perimeter of that node on the diffusion, poly, and metal layers. The lines containing this information look like this:

N *node* *diff-area* *diff-perim* *poly-area* *poly-perim* *metal-area* *metal-perim*

Node is the node name. *Diff-area* through *metal-perim* are the area and perimeter of the diffusion, poly, and metal layers in user defined units. (In addtion the -o option causes transistors with only one terminal to be recorded in the .sim file as a transistor with source connected to drain.)

Setting Options

By default, *mextra* reports locations in CIF units. A more convenient form of units may be specified either in the `.cadrc' file or on the command line. The form of the command line option is:

units *scale*

To set units on the command line use the -u option.

The parameters used to compute node capacitance may be changed by including the following commands in your `.cadrc' file.

areatocap *layer* *value*
perimtocap *layer* *value*

value is atto-farads per square micron for area, and atto-farads per micron for perimeter. *layer* may be "poly", "diff", "metal", or "poly/diff". The threshold for reporting capacitance may set in the `.cadrc' file with the following line.

capthreshold *value*

A negative value sets the threshold to infinity.

Mextra knows of two technologies, NMOS and CMOS p-well. NMOS is assumed by default. To set the technology to CMOS p-well, include the following line in your `.cadrc' file:

```
tech  cmos-pw
```

FILES

```
~cad/lib/extname
~cad/lib/log
~cad/.cadrc
~/.cadrc
/usr/tmp/#mext*
```

ALSO SEE

caesar(cad1), kic(cad1), powest(cad1), cadrc(cad5)

BUGS

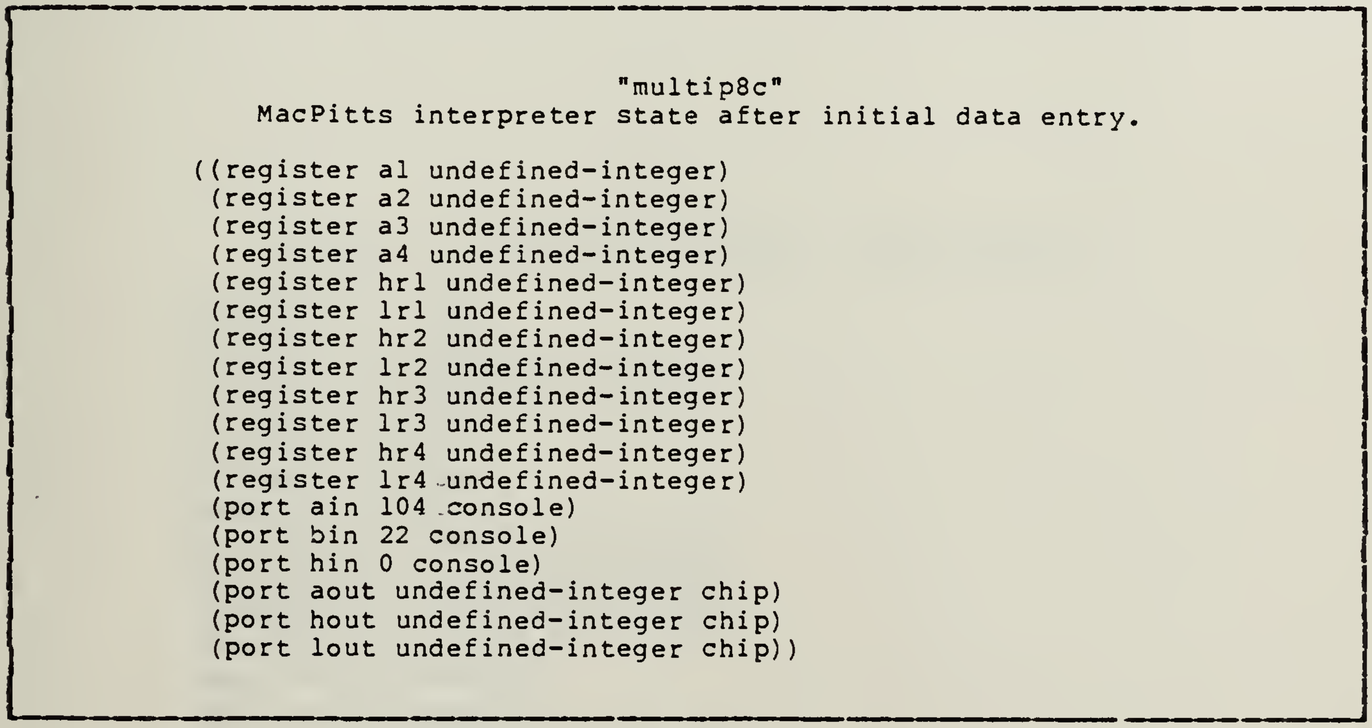
Accepts manhattan simple CIF only. The length/width ratio for unusually shaped transistors may be inacurate. Attributes for funny transistors are not recorded.

APPENDIX D

SIMULATION RESULTS FOR MULTIP8C MULTIPLIER

The first five figures show, in the order that they were produced, .int files from a MacPitts interpreter session using the source file, multip8c.mac.

The last three figures show the terminal output produced by the switch level event simulation program, esim, operating on the node extraction file of the MacPitts layout for multip8c. The node extraction was performed by the mextra program.



```
                         "multip8c"
     MacPitts interpreter state after initial data entry.

((register a1 undefined-integer)
 (register a2 undefined-integer)
 (register a3 undefined-integer)
 (register a4 undefined-integer)
 (register hr1 undefined-integer)
 (register lr1 undefined-integer)
 (register hr2 undefined-integer)
 (register lr2 undefined-integer)
 (register hr3 undefined-integer)
 (register lr3 undefined-integer)
 (register hr4 undefined-integer)
 (register lr4 undefined-integer)
 (port ain 104 console)
 (port bin 22 console)
 (port hin 0 console)
 (port aout undefined-integer chip)
 (port hout undefined-integer chip)
 (port lout undefined-integer chip))
```

Figure D.1 Macpitts Interpreter Results.

```
                        "multip8c"
      MacPitts interpreter state after 1 clock cycle.

((register a1 104)
 (register a2 undefined-integer)
 (register a3 undefined-integer)
 (register a4 undefined-integer)
 (register hr1 0)
 (register lr1 11)
 (register hr2 undefined-integer)
 (register lr2 undefined-integer)
 (register hr3 undefined-integer)
 (register lr3 undefined-integer)
 (register hr4 undefined-integer)
 (register lr4 undefined-integer)
 (port ain 104 console)
 (port bin 22 console)
 (port hin 0 console)
 (port aout undefined-integer chip)
 (port hout undefined-integer chip)
 (port lout undefined-integer chip))
```

```
                        "multip8c"
      Macpitts interpreter state after 2 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 undefined-integer)
 (register a4 undefined-integer)
 (register hr1 0)
 (register lr1 11)
 (register hr2 52)
 (register lr2 5)
 (register hr3 undefined-integer)
 (register lr3 undefined-integer)
 (register hr4 undefined-integer)
 (register lr4 undefined-integer)
 (port ain 104 console)
 (port bin 22 console)
 (port hin 0 console)
 (port aout undefined-integer chip)
 (port hout undefined-integer chip)
 (port lout undefined-integer chip))
```

Figure D.2   MacPitts Interpreter Results, (continued).

```
                         "multip8c"
      Macpitts interpreter state after 3 clock cycles.

((register al 104)
 (register a2 104)
 (register a3 104)
 (register a4 undefined-integer)
 (register hrl 0)
 (register lrl 11)
 (register hr2 52)
 (register lr2 5)
 (register hr3 78)
 (register lr3 2)
 (register hr4 undefined-integer)
 (register lr4 undefined-integer)
 (port ain 104 console)
 (port bin 22 console)
 (port hin 0 console)
 (port aout undefined-integer chip)
 (port hout undefined-integer chip)
 (port lout undefined-integer chip))
```

```
                         "multip8c"
      Macpitts interpreter state after 4 clock cycles.

((register al 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hrl 0)
 (register lrl 11)
 (register hr2 52)
 (register lr2 5)
 (register hr3 78)
 (register lr3 2)
 (register hr4 39)
 (register lr4 1)
 (port ain 104 console)
 (port bin 22 console)
 (port hin 0 console)
 (port aout 104 chip)
 (port hout 39 chip)
 (port lout 1 chip))
```

Figure D.3 MacPitts Interpreter Results, (Continued).

```
                    "multip8c"
    MacPitts interpreter state after 4 clock cycles and
resetting the input ports to the values of the output ports.
This simulates a second chip in cascade with the first.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 0)
 (register lr1 11)
 (register hr2 52)
 (register lr2 5)
 (register hr3 78)
 (register lr3 2)
 (register hr4 39)
 (register lr4 1)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 39 chip)
 (port lout 1 chip))
```

```
                    "multip8c"
      Macpitts interpreter state after 5 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 71)
 (register lr1 128)
 (register hr2 52)
 (register lr2 5)
 (register hr3 78)
 (register lr3 2)
 (register hr4 39)
 (register lr4 1)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 39 chip)
 (port lout 1 chip))
```

Figure D.4 MacPitts Interpreter Results, (Continued).

```
                              "multip8c"
           Macpitts interpreter state after 6 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 71)
 (register lr1 128)
 (register hr2 35)
 (register lr2 192)
 (register hr3 78)
 (register lr3 2)
 (register hr4 39)
 (register lr4 1)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 39 chip)
 (port lout 1 chip))
```

```
                              "multip8c"
           Macpitts interpreter state after 7 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 71)
 (register lr1 128)
 (register hr2 35)
 (register lr2 192)
 (register hr3 17)
 (register lr3 224)
 (register hr4 39)
 (register lr4 1)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 39 chip)
 (port lout 1 chip))
```

Figure D.5 MacPitts Interpreter Results, (Continued).

```
                        "multip8c"
      Macpitts interpreter state after 8 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 71)
 (register lr1 128)
 (register hr2 35)
 (register lr2 192)
 (register hr3 17)
 (register lr3 224)
 (register hr4 8)
 (register lr4 240)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 8 chip)
 (port lout 240 chip))
```

```
                        "multip8c"
      MacPitts interpreter state after 9 clock cycles.

((register a1 104)
 (register a2 104)
 (register a3 104)
 (register a4 104)
 (register hr1 71)
 (register lr1 128)
 (register hr2 35)
 (register lr2 192)
 (register hr3 17)
 (register lr3 224)
 (register hr4 8)
 (register lr4 240)
 (port ain 104 console)
 (port bin 1 console)
 (port hin 39 console)
 (port aout 104 chip)
 (port hout 8 chip)
 (port lout 240 chip))
```

Figure D.6   MacPitts Interpreter Results, (Continued).

```
% esim mul8c.sim mul8c.macro
1612 transistors, 1398 nodes (801 pulled up)
1612 transistors, 1398 nodes (801 pulled up)
sim> s
step took 605 events
clock=XXX
aout=XXXXXXXX
lout=XXXXXXXX
hout=XXXXXXXX
hin=00000000    0
bin=00010110    22
ain=01101000    104
sim> I
initialization took 2119 steps
sim> I
initialization took 0 steps
sim> s
step took 0 events
clock=000       0
aout=11111111   255
lout=11111111   255
hout=11111111   255
hin=00000000    0
bin=00010110    22
ain=01101000    104
sim> c
clock=101       5
aout=11111111   255
lout=01111111   127
hout=01111111   127
hin=00000000    0
bin=00010110    22
ain=01101000    104
cycle took 1433 events
sim> c
clock=101       5
aout=11111111   255
lout=00111111   63
hout=00111111   63
hin=00000000    0
bin=00010110    22
ain=01101000    104
cycle took 1210 events
```

Last line is repeated at top of following page.

Figure D.7 Event Simulation Results.

```
cycle took 1210 events
sim> c
clock=101       5
aout=11111111   255
lout=00011111   31
hout=00011111   31
hin=00000000    0
bin=00010110    22
ain=01101000    104
cycle took 1231 events
sim> c
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00000000    0
bin=00010110    22
ain=01101000    104
cycle took 1139 events
sim> c
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00000000    0
bin=00010110    22
ain=01101000    104
cycle took 1052 events

sim> @ mul8c.macro2
sim> s
step took 177 events
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00100111    39
bin=00000001    1
ain=01101000    104
sim> c
```

Last line is repeated at top of following page.

Figure D.8   Event Simulation Results, (Continued).

```
sim> c
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00100111    39
bin=00000001    1
ain=01101000    104
cycle took 1164 events
sim> c
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00100111    39
bin=00000001    1
ain=01101000    104
cycle took 1154 events
sim> c
clock=101       5
aout=01101000   104
lout=00000001   1
hout=00100111   39
hin=00100111    39
bin=00000001    1
ain=01101000    104
cycle took 1131 events
sim> c
clock=101       5
aout=01101000   104
lout=11110000   240
hout=00001000   8
hin=00100111    39
bin=00000001    1
ain=01101000    104
cycle took 1123 events
sim> c
clock=101       5
aout=01101000   104
lout=11110000   240
hout=00001000   8
hin=00100111    39
bin=00000001    1
ain=01101000    104
cycle took 1052 events
sim> q
%
```

Figure D.9   Event Simulation Results, (Continued).

APPENDIX E

LAYOUT PHOTOGRAPHS

Exposure data:

Display: AED 767 Color Graphics Terminal
Weston Light Value: 7.5 to 8.5
Camera: Pentax SLR, Tripod Mounted
Film: Tri-X, ASA 400
Lens-to-screen distance: 4 feet
Lens: 85 mm, f1.9
Lens Opening: f16
Shutter Speed: 1 second

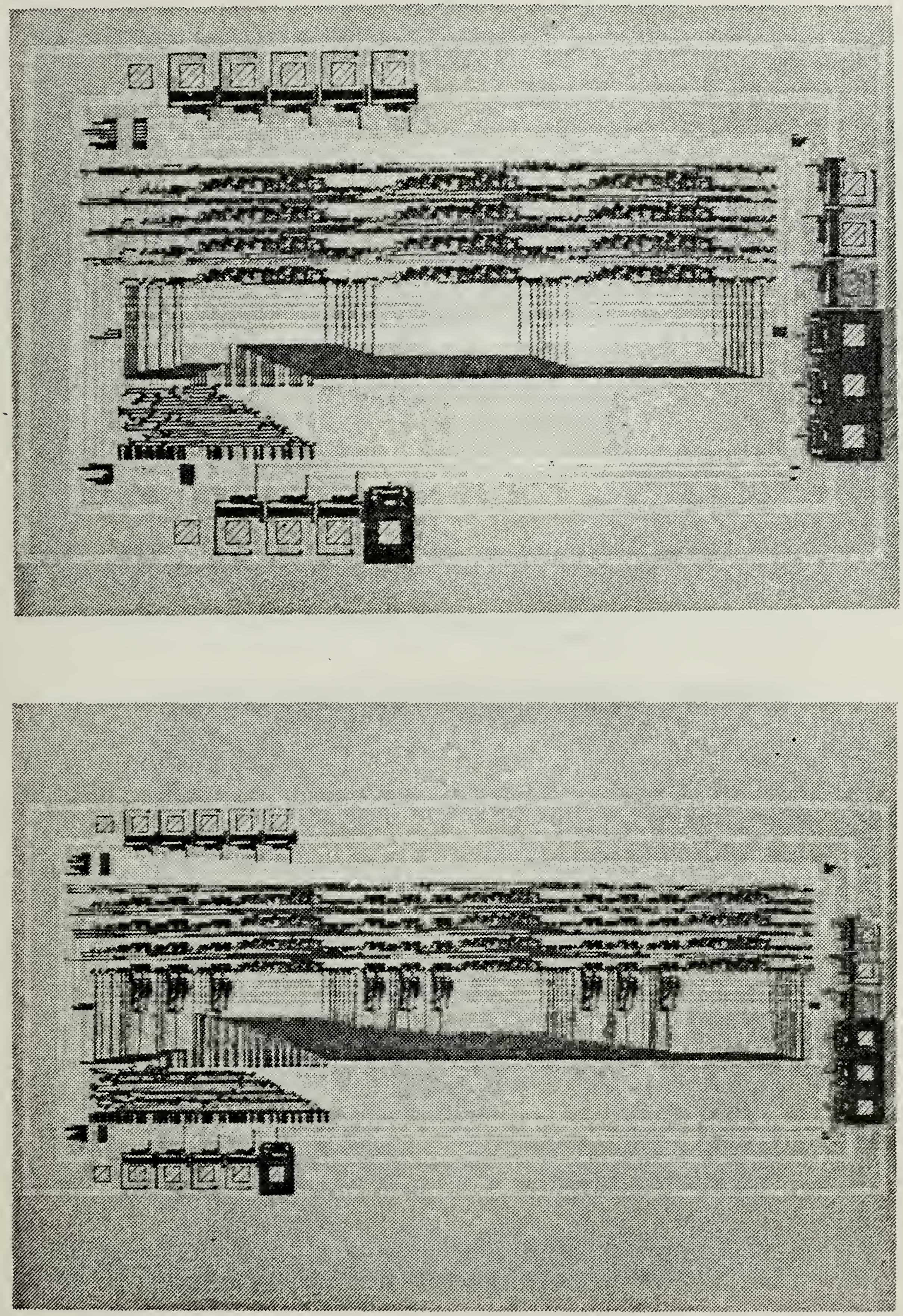


Figure E.1 multic (top), multip (bot).

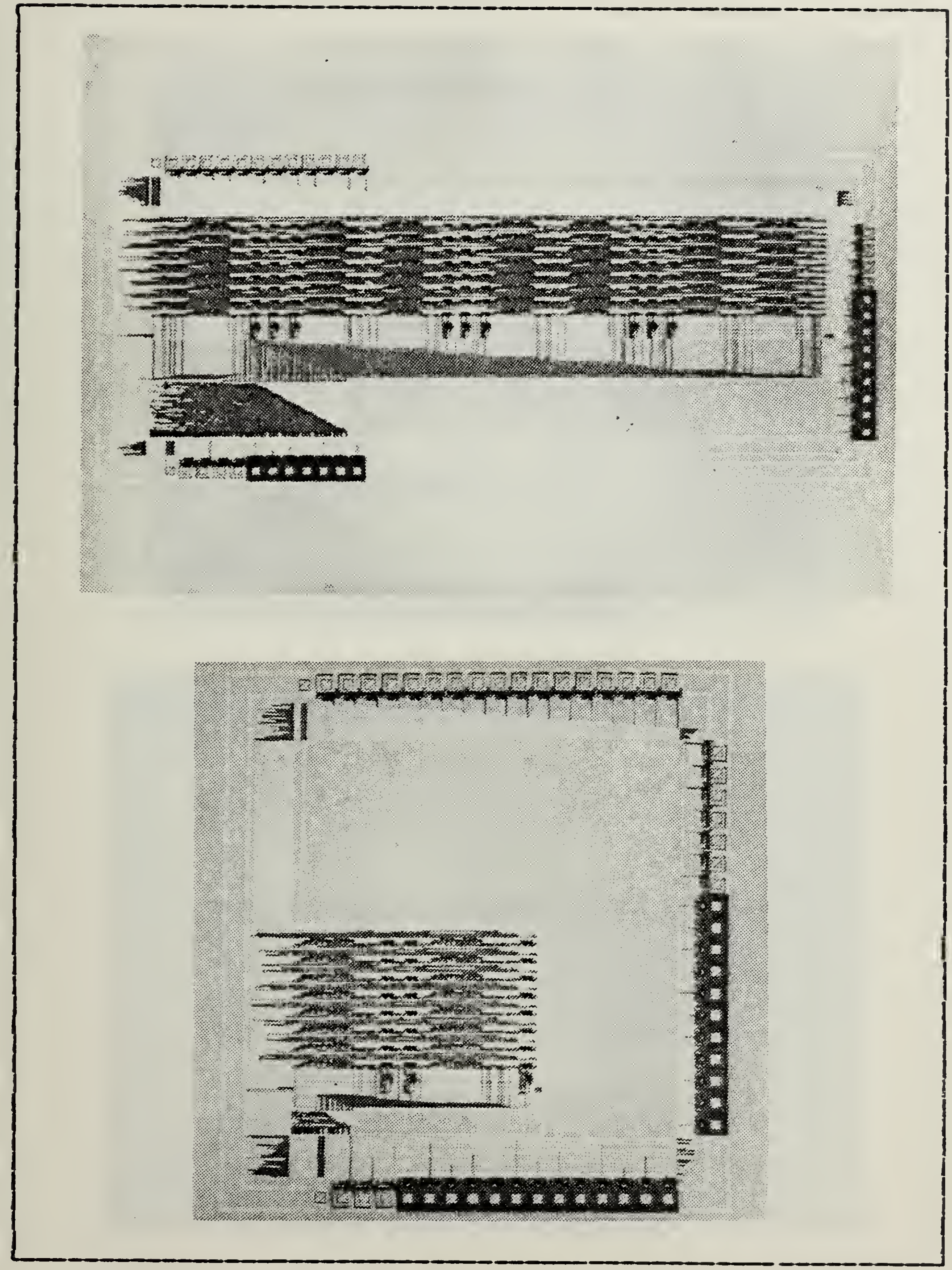


Figure E.2 multip8 (top), multip8a (bot).

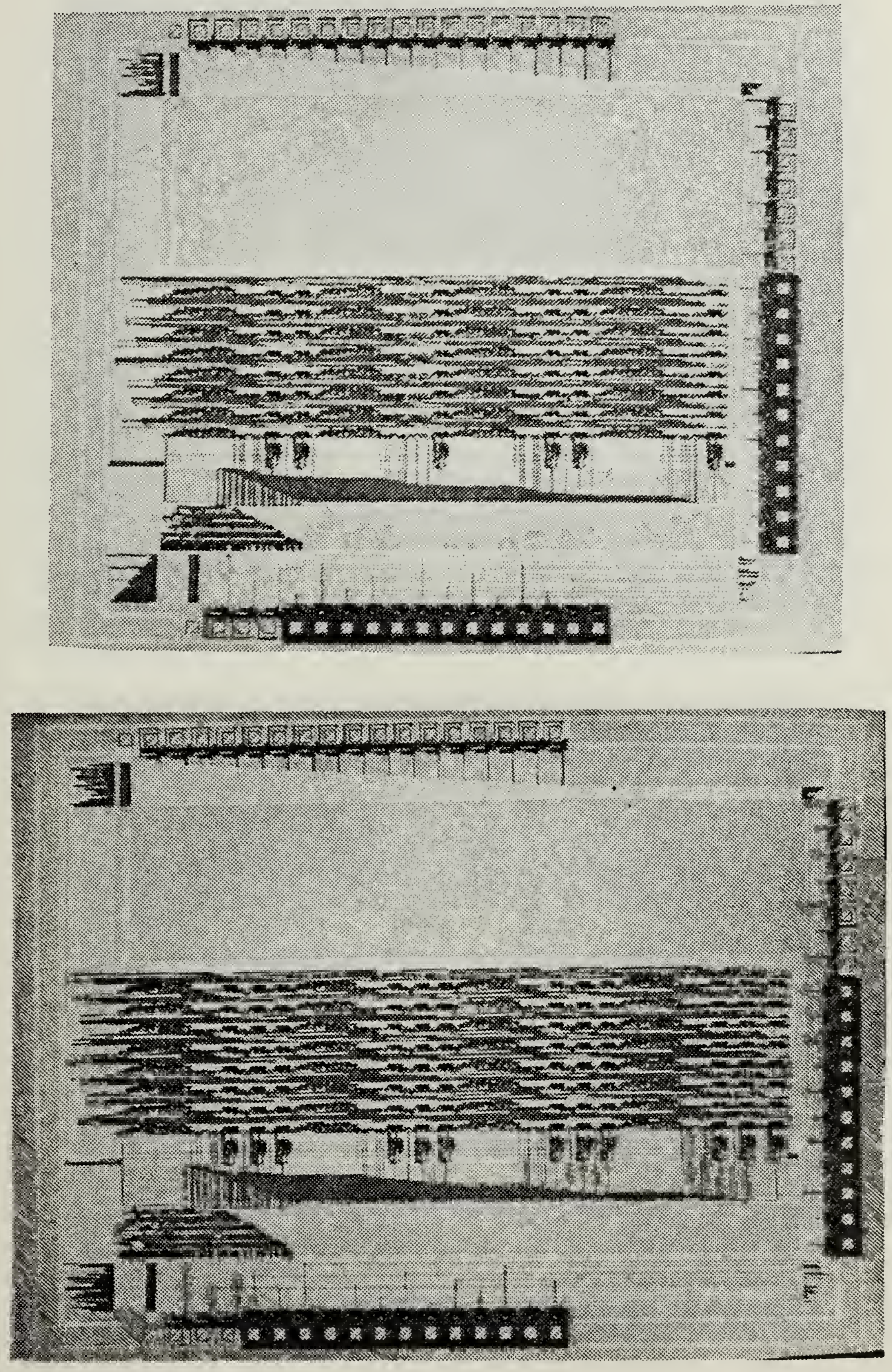


Figure E.3 multip8b (top), multip8c5 (bot).

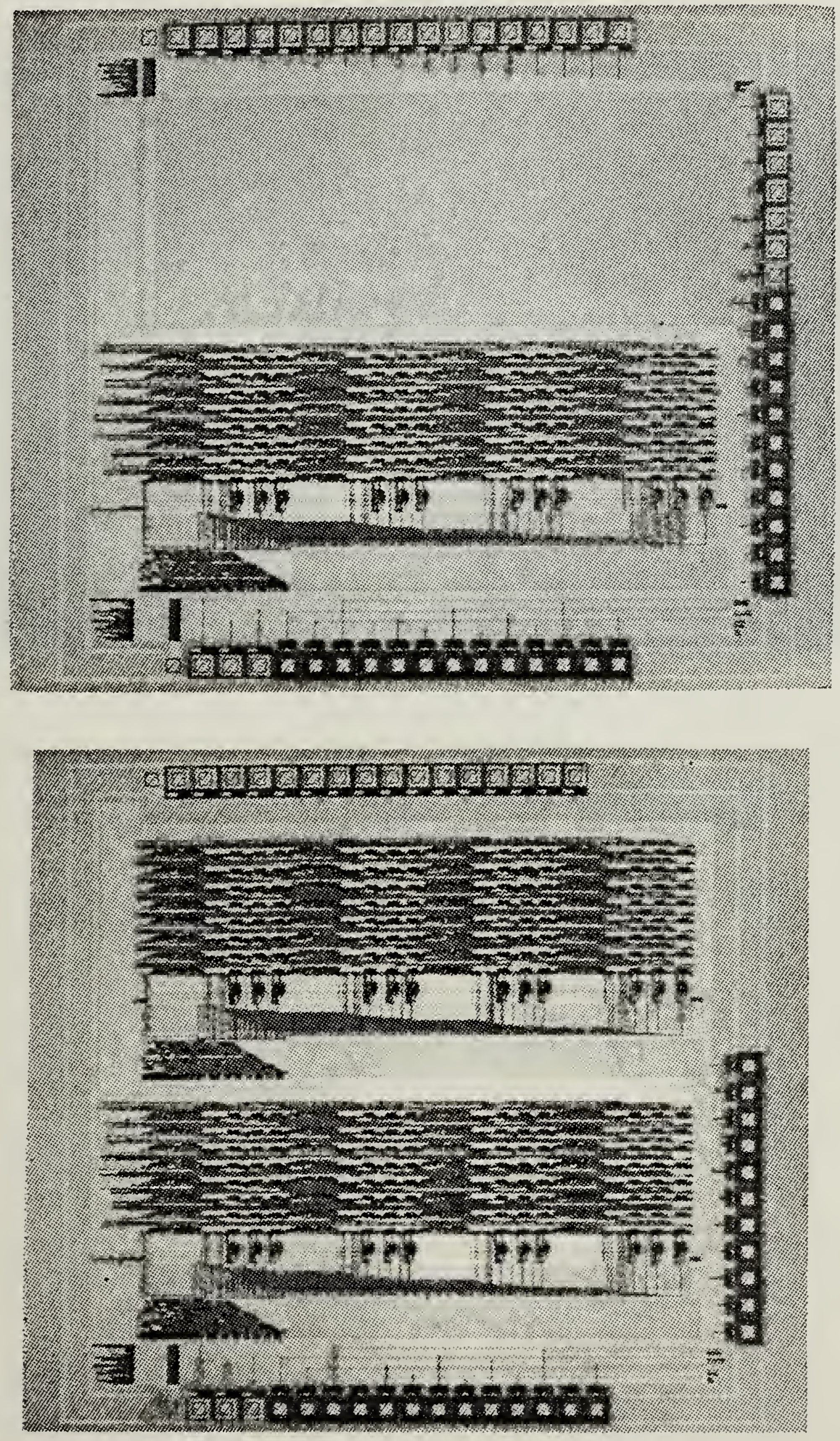


Figure E.4 multip8c4 (top), multip8c4d (bot).

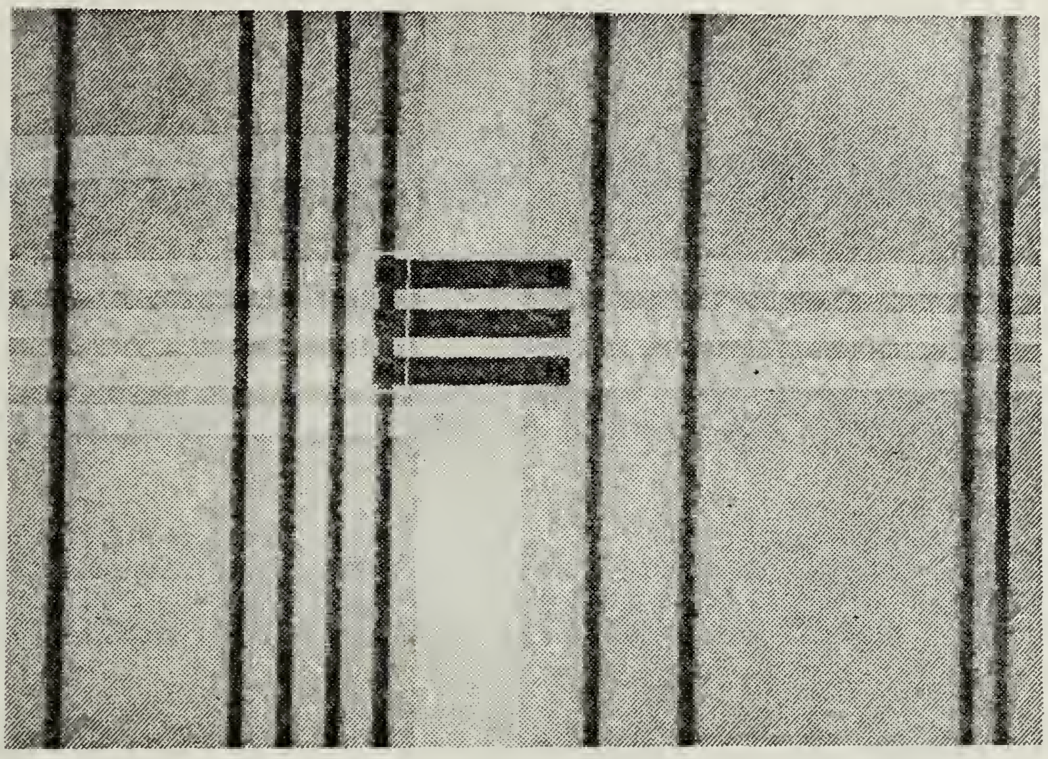


Figure E.5 Layout Errors in kchip2.

LIST OF REFERENCES

1. Conradi, J. R. and Hauenstein, B. R., *VLSI Design of a Very Fast Pipelined Carry Look Ahead Adder*, Master's Thesis, Naval Postgraduate School, September 1983.

2. Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

3. Southard, J. R., "MacPItts: An Approach to Silicon Compilation," *Computer*, v. 16, pp. 74-82, December 1983.

4. Siskind, J. M, Southard, J. R. and Crouch, K. W., "Generating Custom High Performance VLSI designs from Succinct Algorithmic Descriptions," *Proceedings, Conference on Advanced Research in VLSI*, pp. 28-40, 1981.

5. Wallich, P., "On the Horizon: Fast Chips Quickly," *IEEE Spectrum*, v. 21, pp. 28-34, March 1984.

6. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, *Editing VLSI Circuits With Caesar*, by J. Ousterhout, pp. 1-21, March 22, 1983.

7. Werner, J., "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?" *VLSI Design*, v. 3, pp. 46-52, September/October 1982.

8. The Microelectronics Center of North Carolina Technical Report 83-06, *Silicon Compilers: A Critical Survey*, by R. R. Gross, 31 May 1983.

9. Werner, J., "Progress Toward the 'Ideal' Silicon Compiler, Part II: the Layout Problem," *VLSI Design*, v.4, pp. 78-81, October 1983.

10. Johansen, D., "Bristle Blocks: A Silicon Compiler," *Sixteenth Design Automation Conference Proceedings*, pp. 310-313, IEEE Computer Society, 1979.

11. Bergman, N., "A Case Study of the F. I. R. S. T. Silicon Compiler," *Proceedings, Third CalTech Conference on VLSI*, Pasadena, Calif., pp. 413-430, March 1983.

12. Ayres, R. F., *VLSI: Silicon Compilation and the Art of Automatic Microchip Design*, pp. 3-5, Prentice-Hall, 1983.

13. University of California at Berkeley, *The Franz Lisp Manual*, by J. K. Foderaro, K. L. Sklower, and Kevin Layer, June 1983.

14. Lincoln Laboratory, Massachusetts Institute of Technology Project Report RVLSI-5, *L5 Users Guide*, by K. W. Crouch, p. 19, 7 March 1984.

15. Weinberger, A., "Large Scale Integration of MOS Complex Logic: A Layout Method," *IEEE Journal of Solid State Circuits*, v. sc-2, pp. 182-190 December 1967.

16. Lincoln Laboratory, Massachusetts Institute of Technology Project Report RVLSI-3, *An Introduction to MacPitts*, by J. R. Southard, 10 February 1983.

17. Loomis, H. Jr. and Sinha, B., "High Speed Recursive Digital Filter Realization," pre-print of a paper to appear in *Circuits, Systems and Signal Processing*.

18. Kung, H. T. and Leiserson, C. E., "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings, 1978*, Society for Industrial and Applied Mathematics, pp. 256-282, 1979.

19. Kelly, G. I., *A Custom Integrated Circuit for Connecting Multiple Processing Elements Using a Baseline Network*, paper presented at the Seventeenth Annual Asilomar Conference on Circuits and Systems, Pacific Grove, California, 1 November 1983.

20. Newkirk, J. A. and Mathews, R., *The VLSI Designer's Library*, Addison-Wesley, 1983.

BIBLIOGRAPHY

Bourne, S. R., *The UNIX System*, Addison-Wesley, 1983.

Feldman, J. A. and Beauchemin, E. J., "A Custom IC for Automatic Gain Control in LPC Vocoders," *Proceedings of ICASSP '83*, Boston, Mass, April 1983.

Fox, J. R., "The MacPItts Silicon Compiler: A View From the Telecommunications Industry," *VLSI Design*, v. 4, May/June 1983.

Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, 1979.

Kogge, P. M., *The Architecture of Pipelined Computers*, Hemisphere, 1981.

Srini, V. P., "Test Generations from MacPitts Designs," *Proceedings of the Internations Conference on Computer Design*, November 1983.

Wallich, P., "Technology '83: Automation (Design/Manufacturing)," *IEEE Spectrum*, v. 20, January 1983.

Werner, J., "Progress Toward the 'Ideal' Silicon Compiler, Part I: the Front End," *VLSI Design*, v. 4, September 1983.

Winston, P. H. and Berthold, K. P. H., *LISP*, Addison-Wesley, 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Superintendent Attn: Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
2.	Dr. Donald Kirk Code 62KI Naval Postgraduate School Monterey, CA 93943	5
3.	Prof. Robert Strum Code 62ST Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. H. H. Loomis Code 62LM Naval Postgraudate School Monterey, CA 93943	1
5.	Mr. Albert Wong Code 52 Naval Postgraduate School Monterey, CA 93943	1
6.	Chairman, ECE Department Code 62 Naval Postgraduate School Monterey, CA 93943	1
7.	LCDR Dennis J. Carlson Fleet Air Reconnaissance Squadron Three FPO San Francisco, CA 96601	1
8.	LT Joseph R. Conradi 5293 Iroquois Avenue Ewa Beach, HI 96706	1
9.	Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
10.	Dr. Gerry L. Kelly Department of ECE University of Kansas Lawrence, Kansas 66045	1
11.	Mr. A. E. Hastings Dept. 81-60, Bldg. 157 Lockheed Missle and Space Company P. O. Box 504 Sunnyvale, CA 94086	1
12.	Mr. G. S. Ogden Dept. 81-61, Bldg. 157 Lockheed Missle and Space Company P. O. Box 504 Sunnyvale, CA 94086	1

13. Dr. Antun Domic, B-347 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, MA 02173-0073

14. Mr. Peter Blankenship 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, MA 02173-0073

208296

Thesis
C2328 Carlson
c.1 Application of a silicon compiler to VLSI design of digital pipelined multipliers.